华章科技

经典畅销书系“深入理解Android”系列Framework卷完结篇，数十万Android开发工程师翘首以盼

移动开发

从源代码层面全面、详细剖析了Android 框架UI系统的实现原理和工作机制，以及优秀代码的设计思想，填补市场空白



张大伟◎著

Understanding Android Internals: Volume III

深入理解Android

卷 III



机械工业出版社
China Machine Press

目錄

深入理解Android 卷III

作者：张大伟

来源：Innost的专栏

推荐序

回顾

今天是一个值得高兴的日子。历经两年多的艰苦奋斗，张大伟同学的这本著作，同时也是隶属“深入理解Android”系列三卷中的最后一卷终于完成了。从2011年我和华章公司的杨福川编辑一起开创这一迄今为止国内Android技术书籍市场上唯一一套兼具广度和深度的“深入理解Android”系列书籍算起，四个年头已经过去。在这四年中，本套书籍的作者们和出版社的编辑们共同奋斗，成果斐然：

- 2011年9月《深入理解Android 卷I》发布。
- 2012年8月《深入理解Android 卷II》发布。
- 2013年1月，本系列的第一本专题卷《深入理解Android：Telephony原理剖析与最佳实践》发布，作者是杨青平。
- 2014年4月，本系列的第二本专题卷《深入理解Android：Wi-Fi，NFC和GPS卷》发布。
- 2015年，《深入理解Android 卷III》发布，作者即是本书的主人公张大伟。
- 2015年及以后，我们还要发布深入理解Android系列书籍中的WebKit专题卷、自动化测试卷、蓝牙专题卷等。

从技术层面来说，本书填补了深入理解Android Framework卷中的一个主要空白，即Android Framework中和UI相关的部分。在一个特别讲究颜值的时代，本书分析了Android 4.2中WindowManagerService、ViewRoot、Input系统、StatusBar、Wallpaper等重要“颜值绘制/处理”模块。虽然在写书的两年中，Android版本已经从4.2进化到M，但“面虽新，神依在”。所以，我可以很负责任地说，对那些掌握了本书精髓的读者而言，即使Android未来升级到了X，那也只不过是换了一个马甲罢了。

展望

我在卷II中曾经详细阐述过“深入理解Android”这一系列的路线图（http://blog.csdn.net/innost/article/details/7648869），这里在展现一下当时的情况。



本套丛书大体分为应用部分、Framework部分、专题部分和内核部分。

1）应用部分。这部分拟以Android源码中自带的那些应用程序为分析目标，充分展示Google在自家SDK平台上做应用开发的深厚功力。这些应用包括Contacts、Gallery2、Mms、Browser等，它们的分析难度都不可小觑。通过对这些系出名门的应用的分析，我们希望读者不仅能把握商业级应用程序开发的精髓，而能更精熟地掌握Android应用开发的各种技能。

2）Framework部分。关注Android的框架，包括三本书。

- 卷I：以Native层Framework模块为分析对象。知识点包括init、binder、zygote、jni、Message和Handler、audio系统、surface系统、vold、rild和mediascanner。本书已于2011年9月出版，虽然是基于Android 2.2，读者如若扎实地掌握并理解了其中的内容，那么以后再研究2.3或4.0版本中对应的模块，也是轻而易举之事了。
- 卷II和卷III：以Java层Framework模块为分析对象。卷II基于4.0.1版，包括UI相关服务和Window系统之外的一些重要服务，如PackageManagerService、ActivityManagerService、PowerManagerService、ContentService、ContentProvider等。而的卷III将以输入系统、WindowManagerService、UI相关服务为主要目标。

Framework部分所包括的这3本书的目的是让读者对整个Android系统有较大广度、一定深度的认识，这有益于读者能构建一个更为完整的Android系统知识结构。应当指出，这3本书不可能覆盖Android Framework中的所有知识点。因此，尚需读者在此基础上，结合不同需求，进行进一步的深入研究。

3）专题部分。旨在帮助读者沿着Android平台中的某一些专业方向，进行深度挖掘，拟规划如下专题：

- Telephony专题，涵盖SystemServer中相关的通信服务、rild、短信、电话等模块。
- 多媒体专题，涵盖MultiMedia相关的模块，包括Stagefright、OMX等。另外，我们也打算引入开源世界中最流行的一些编解码引擎和播放引擎作为分析对象。
- 浏览器和Webkit专题，该专题难度非常大，但其重要性却不言而喻。

- Dalvik虚拟机专题，该专题希望对Dalvik进行一番深度研究，涉及面包括Java虚拟机的实现、Android的一些特殊定制等内容。现在来看，Dalvik已经被ART替换，所以这本书的目标就应该是ART虚拟机专题了。
- Android系统安全专题，该专题的目标是，分析Android系统上提供的安全方面的控制机制。另外，Linux平台上的一些常用安全机制（例如，文件系统加密等）也是本书所要考虑的。这套安全专题我已经在自己的博客[1]上写了部分内容，包括Java Security、设备加密等。
- UI/UE设计以及心理学专题，：该专题希望能提供一些心理学方面的指导以及具体的UI/UE设计方面的指南以帮助开发人员开发出更美、更体贴和更方便的应用。

专题部分隐含着的一个极为重要的宗旨：即基于Android，而高于Android。换言之，这些书籍虽都以Android为切入点，但我们更希望读者学到的知识、掌握的技术却不局限于Android平台。

4） 内核部分。这部分图书拟以Linux内核为主。虽然这方面的经典教材非常多，但要么是诸如《Linux内核情景分析》之类的鸿篇巨帙，要么是类似《Linux内核设计与实现》，内容过于简洁。另外，现有书籍使用的内核源码都已比较陈旧。为此，我们希望能有一本难度适中、知识面较广、深度适宜的书籍。

今天，正是由于大伟的努力，我们的Framework部分得以完美收官。高兴的同时，我们认为前路依然艰辛。在此，我和福川兄再次诚挚邀请国内外有热情，愿分享、有责任心的兄弟姐妹们来一起继续发扬光大“深入理解Android”这一系列书籍。

还是杨澜的那句话，“原来我只佩服成功的人，现在我更尊敬那些正在努力的人”。让我们一起成为被尊敬的人吧！

轶事

我和大伟相知相识的过程还颇有点意思。

那时我们都在中科创达工作，有一次，我们俩要一起重构一个和音频相关的解码模块。当时我噼噼啪啪把几段和多线程相关的同步代码块改写后，引起了大伟的强烈质疑。在质疑（challenge）和争论（argue）的过程中，我发现大伟思路清晰，技术能力较强，是一个不可多得的好苗子，便有意交往。虽然吵得很激烈，不过最终实践的结果是这次改写还是比较成功的，这使得我赢得了大伟的信任。

交手过后，我们便成了好兄弟。2012年夏天，我和大伟被派遣到上海高通公司OnSite。当时我刚完成了卷II的撰写，同时也在思考很多读者提出的一个问题，即什么时候能详细分析一下AndroidFramework UI部分。古语云“书如其人”，对于我这样一个对颜值不是很讲究的人来说，写这本书肯定不是最合适的。因为我觉得这边书的作者需要耐心、细心、同时还需要一定审美观。在我认识的技术能力较强的兄弟们中，大伟无疑是最适合撰写本书的人选。

当然，对于一个从未写过书籍的人而言，写书这样的重任最初还是让大伟觉得紧张，感觉没有信心。所以，我和大伟一起签的合同，让他觉得自己不是孤身作战。另外，在一些技术难点上，我会编写一些小例子，让大伟去完善，并以这些例子为出发点来分析Framework的实现。最后，大伟凭借自己的的天分和努力，很快就从一个跟随者变成了这本书的主导者和唯一作者。

在本书的审稿过程中，我很欣慰得发现这本书细节深入、知识全面，是一本诚意之作。在此，我个人非常感谢大伟的努力，这本书了却了我多年的一桩心愿。

我曾经很羡慕那些有战友之情的士兵们。在和平年代的今天，我觉得我和大伟，福川，杨青平等作者、编辑都曾为了共同一个目标一起努力过，奋斗过，我们之间的感情应该能够媲美战友之情吧。

邓凡平

2015-7-5

> [1] 我的博客地址：。

第1章 开发环境部署

本章主要内容：

- 介绍编译环境的搭建以及如何利用Eclipse调试SystemServer进程。

1.2 搭建开发环境

本节将讨论Android源码下载、Eclipse开发环境，以及如何调试SystemServer进程等相关知识。

1.2.1 下载源码

Android 2.3以后，Google官方推荐使用64位的操作系统来编译源码。所以，读者要先安装64位的OS。笔者推荐的操作系统是Ubuntu 10.04 X86-64版。

笔者不打算过多讨论Android源码下载的步骤，原因是：这是一个需要读者动手操作的过程，而看着电脑屏幕操作比看书后输入大串的字符串的效率要高很多。

基于上面这个原因，这里笔者向读者提供一个官方说明文档，地址是 http://source.android.com/source/downloading.html。该网页中有详细的代码下载步骤，读者只要执行简单的复制/粘贴操作即可动手实验。如图图1-3所示为该网页的截图。

Downloading the Source Tree

Installing Repo

Repo is a tool that makes it easier to work with Git in the context of Android. For more information about Repo, see Version Control.

To install, initialize, and configure Repo, follow these steps:

- Make sure you have a bin/ directory in your home directory, and that it is included in your path:

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
```

- Download the Repo script and ensure it is executable:

```
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

- The SHA-1 checksum for repo is 29ba4221d4fccdfa8d87931cd73466fdc24040b5

Initializing a Repo client

After installing Repo, set up your client to access the android source repository:

- Create an empty directory to hold your working files. If you're using MacOS, this has to be on a case-sensitive filesystem. Give it any name you like:

```
$ mkdir WORKING_DIRECTORY
$ cd WORKING_DIRECTORY
```

- Run repo init to bring down the latest version of Repo with all its most recent bug fixes. You must specify a URL for the manifest, which specifies where the various repositories included in the Android source will be placed within your working directory.

```
$ repo init -u https://android.googlesource.com/platform/manifest
```

To check out a branch other than "master", specify it with -b:

```
$ repo init -u https://android.googlesource.com/platform/manifest -b android-4.0.1_r1
```

图1-3 Android代码下载网页示意图

注意 读者要选择下载Android 4.0.1的代码。虽然最新的Android 4.0.3版本从版本号上看变化不大，但实际代码却有较大变化。

1.2.2 编译源码

1. 部署JDK

Android 2.3及以后版本的代码编译都需要使用JDK1.6，所以首先要做的是下载JDK1.6。下载网址是http://www.oracle.com/technetwork/java/javasebusiness/downloads/[java-archive-downloads-javase6-419409.html](http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html)。笔者下载的文件是jdk-6u27-linux-x64.bin。把它放到一个目录中，比如将其放到/mnt/hgfs/E目录下，然后在这个目录中执行：

```
./ jdk-6u27-linux-x64.bin  #执行这个文件
```

这个命令的作用其实就是解压。解压后的结果在/mnt/hgfs/E/jdk1.6.0_27目录中。有了JDK后，还需要设置~/.bashrc文件。在该文件末尾添加如图1-4所示的几行语句：

```
export JAVA_HOME=/mnt/hgfs/E/jdk1.6.0_27
export JRE_HOME=JAVA_HOME/jre
export CLASSPATH=$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH
```


图1-4 Java环境部署示意图

重新登录系统后，Java环境就添加到系统中了。

2. 编译源码

源码编译的步骤非常简单。我们在卷I也详细介绍编译方法。不过本书要求读者必须先编译整个系统，步骤如下：

- 执行source build/envsetup.sh命令。该命令将导入Android编译环境。
- 输入choosecombo并执行，它是envsetup.sh中定义的一个函数。在执行过程中，分别选择release、generic、eng即可。最终屏幕输出如图1-5所示。

```
root@innost:/[illegible]/work-branches/Android-4.0# choosecombo
Build type choices are:
     1. release
     2. debug

Which would you like? [1]


Which product would you like? [full] generic


Variant choices are:
     1. user
     2. userdebug
     3. eng
Which would you like? [eng]

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.1
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=ITL41D
============================================
```


图1-5 编译设置效果图

- 执行make命令以编译整个系统。编译时间由机器配置决定。笔者的4核4GB机器的编译时间大概在2小时左右。

1.2.3 利用Eclipse调试SystemServer

本节将介绍如何利用Eclipse来调试Android Java Framework的核心进程SystemServer。

1. 配置Eclipse

首先要下载Android SDK。下载地址为http://developer.android.com/sdk/index.html。在Linux环境下，该网站截图如图1-6所示。

Platform	Package	Size	MD5 Checksum
Windows	android-sdk_r18-windows.zip	37448775 bytes	bfbfdf8b2d0fdecc2a621544d706fa98
	installer_r18-windows.exe (Recommended)	37456234 bytes	48b1fe7b431afe6b9c8a992bf75dd898
Mac OS X (intel)	android-sdk_r18-macosx.zip	33903758 bytes	8328e8a5531c9d6f6f1a0261cb97af36
Linux (i386)	android-sdk_r18-linux.tgz	29731463 bytes	6cd716d0e04624b865ffed3c25b3485c

图1-6 AndroidSDK下载网页示意图

笔者下载的是Linux系统上的SDK。解压后的位置在/thunderst/disk/anroid/android-sdk-linux_x86下。

然后要为Eclipse安装ADT插件（Android Development Tools），步骤如下。

- 单击Eclipse菜单栏Help->Install New Software，输入Android ADT下载地址：https://dl-ssl.google.com/android/eclipse/，然后安装其中的所有组件，并重启Eclipse。
- 单击Eclipse菜单栏Preferences->Android一栏，在右边的SDK Location中输入刚才解压SDK后得到的目录，即笔者设置的/thunderst/disk/anroid/android-sdk-linux_x86，最终结果如图1-7所示。

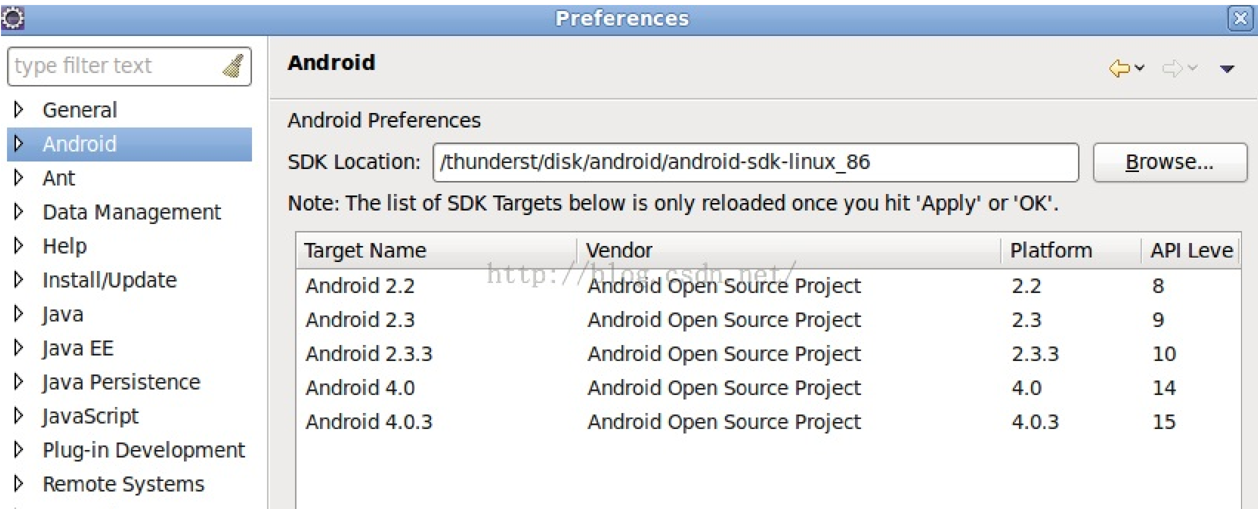


图1-7 SDK安装示意图

- 单击Eclipse菜单栏Window->Android SDK Manager，弹出一个对话框，如图1-8所示。

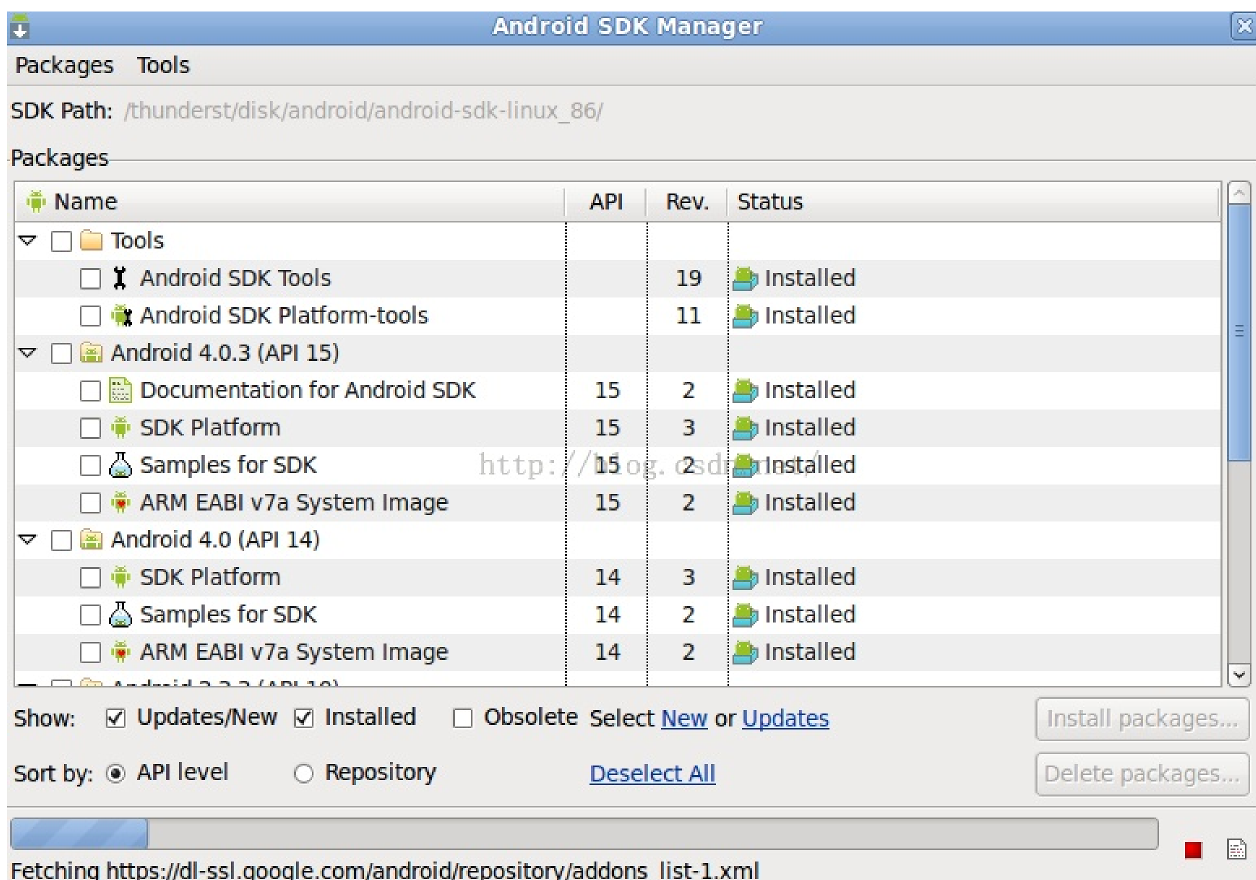


图1-8 Android SDK Manager对话框

在图1-8中选择下载其中的Tools和对应版本的SDK文档、镜像文件、开发包等。有条件的读者可以将Android 4.0.3和4.0对应的内容及Tools全部下载过来。

2. 使用Coffee Bytes Java插件

Coffee BytesJava是Eclipse的一个插件，用于对代码进行折叠，其功能比Eclipse自带的代码折叠功能强大多了。对于大段代码的研究，该插件的作用功不可没。此插件的安装和配置步骤如下：

- 单击Eclipse菜单栏Help->Install New Software，在弹出的对话框中输入http://eclipse.realjenius.com/update-site，选择安装这个插件即可。
- 单击Eclipse菜单栏Window->Preference，在左上角输入Folding进行搜索，结果如图1-9所示。

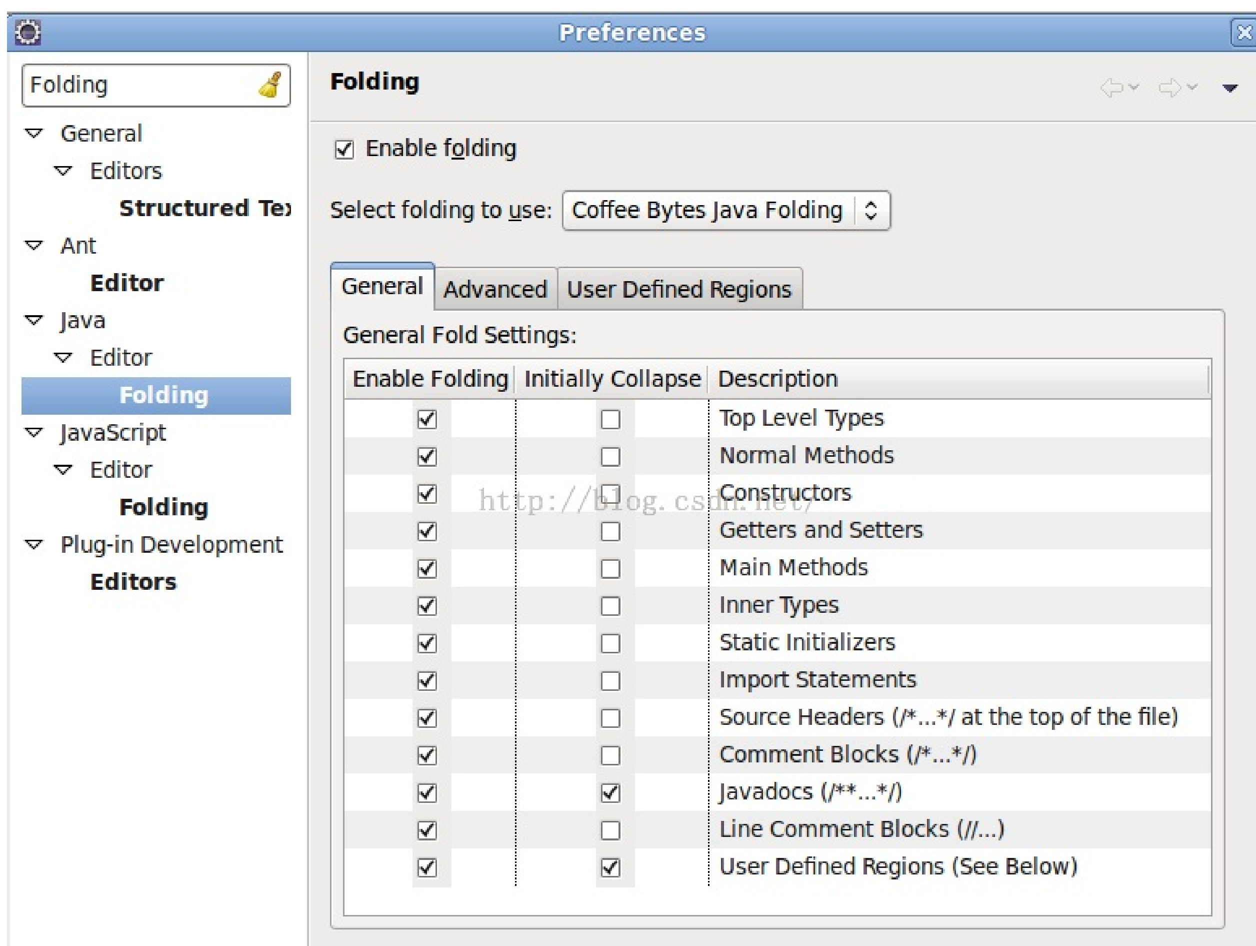


图1-9 Coffee Bytes Java 插件配置

在图1-9所示的对话框中，要先打开Enablefolding选项，然后从Selectfolding to use框中选择CoffeeBytes Java Folding。图1-9右下部分的勾选框用来配置此插件的代码折叠功能。读者不妨按照图1-9所示来配置它。使用该插件后的示意图如图1-10所示。

如果不关心else if分支，就可以把这段代码折叠起来

```
            boolean onlyCore = false;
            if (ENCRYPTING_STATE.equals(cryptState)) {
                Slog.w(TAG, "Detected encryption in progress - only parsing core apps");
                onlyCore = true;
            } else if (ENCRYPTED_STATE.equals(cryptState)) {
            }

            pm = PackageManagerService.main(context,
                    factoryTest != SystemServer.FACTORY_TEST_OFF,
                    onlyCore);
            boolean firstBoot = false;
            try {
                firstBoot = pm.isFirstBoot();
            } catch (RemoteException e) {
            }
```

图1-10 Coffee Bytes Java插件的使用示例

从图1-10中可看到，使用该插件后，基本上代码中所有分支都可以折叠起来，该功能将帮助开发人员集中精力关注自己所关心的分支。

3. 导入Android源码

注意，这一步必须编译完整个Android源码才可以实施，步骤如下：

- 复制Android源码目录/development/ide/eclipse/.classpath到Android源码根目录。
- 打开Andriod源码根目录下的.classpath文件。该文件是供Eclipse使用的，其中保存的是源码目录中各个模块的路径。由于我们只关心Framework相关的模块，因此可以把一些不是Framework的目录从该文件中注释掉。同时，去掉不必要的模块也可加快Android源码导入速度。图1-11所示为该文件的部分内容。

```
	<classpathentry kind="src" path="frameworks/base/cmds/am/src"/>
	<classpathentry kind="src" path="frameworks/base/cmds/input/src"/>
	<classpathentry kind="src" path="frameworks/base/cmds/pm/src"/>
	<classpathentry kind="src" path="frameworks/base/cmds/svc/src"/>
	<classpathentry kind="src" path="frameworks/base/core/java"/>
	<classpathentry kind="src" path="frameworks/base/drm/java"/>
	<classpathentry kind="src" path="frameworks/base/graphics/java"/>
	<classpathentry kind="src" path="frameworks/base/icu4j/java"/>
	<classpathentry kind="src" path="frameworks/base/keystore/java"/>
	<classpathentry kind="src" path="frameworks/base/location/java"/>
	<classpathentry kind="src" path="frameworks/base/media/java"/>
	<classpathentry kind="src" path="frameworks/base/obex"/>
	<classpathentry kind="src" path="frameworks/base/opengl/java"/>
	<classpathentry kind="src" path="frameworks/base/packages/SettingsProvider/src"/>
	<classpathentry kind="src" path="frameworks/base/packages/SystemUI/src"/>
	<classpathentry kind="src" path="frameworks/base/policy/src"/>
	<classpathentry kind="src" path="frameworks/base/sax/java"/>
	<classpathentry kind="src" path="frameworks/base/services/java"/>
	<classpathentry kind="src" path="frameworks/base/telephony/java"/>
	<classpathentry kind="src" path="frameworks/base/test-runner/src"/>
	<classpathentry kind="src" path="frameworks/base/voip/java"/>
	<classpathentry kind="src" path="frameworks/base/wifi/java"/>
	<classpathentry kind="src" path="frameworks/ex/carousel/java"/>
	<classpathentry kind="src" path="frameworks/ex/chips/src"/>
	<classpathentry kind="src" path="frameworks/ex/common/java"/>
	<classpathentry kind="src" path="frameworks/ex/variablespeed/src"/>
	<classpathentry kind="src" path="frameworks/opt/calendar/src"/>
	<classpathentry kind="src" path="frameworks/opt/vcard/java"/>
```


图1-11.classpath文件部分文件内容

另外，一些不必要的模块会导致后续在Eclipse中Android源码编译失败。笔者共享了一个.classpath文件，读者可从http://download.csdn.net/detail/innost/4247578下载并直接使用。

单击Eclipse菜单栏New->Java Project，弹出如图1-12所示的对话框。设置Location为Android4.0源码所在路径。

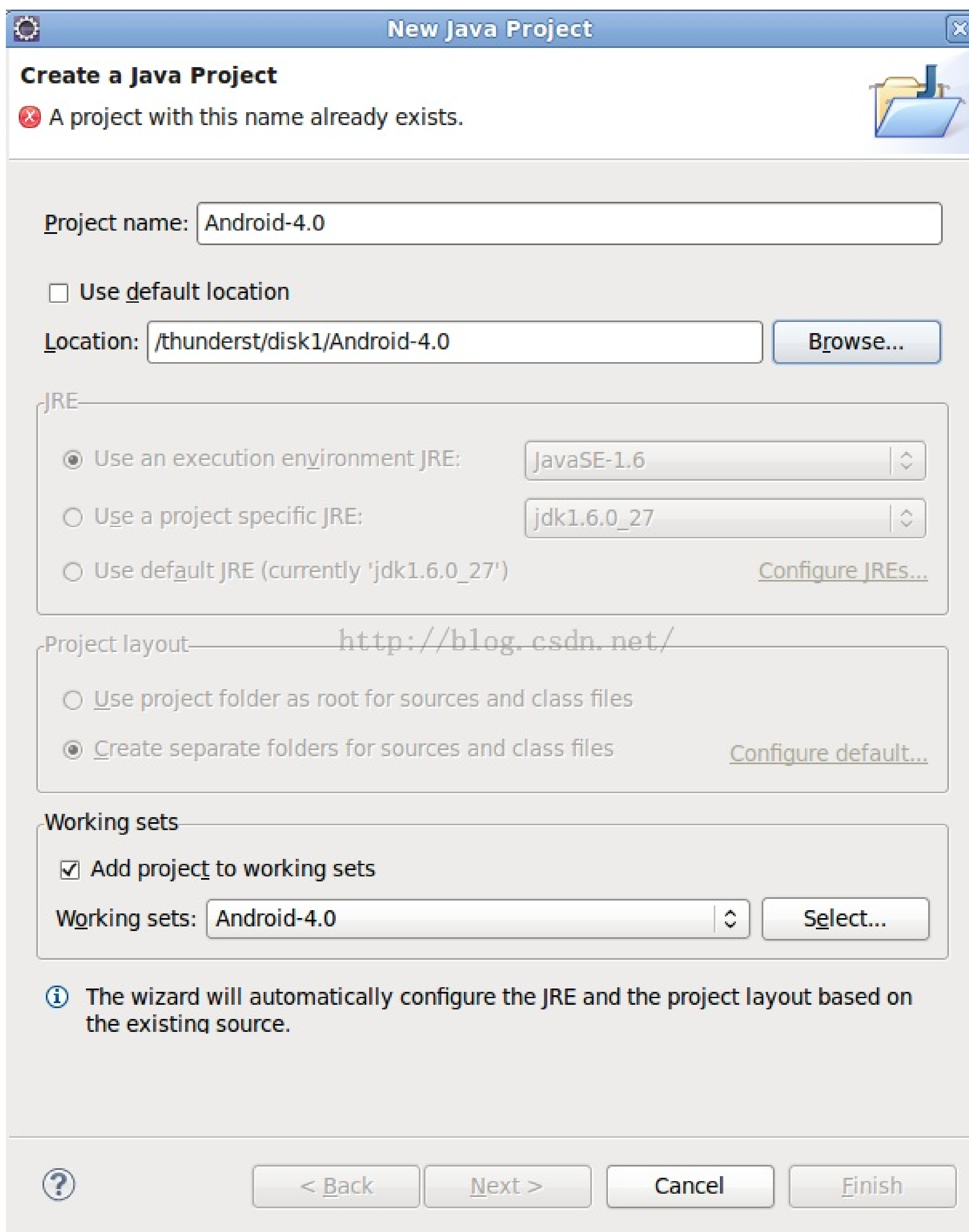


图1-12 导入Android源码示意图

由于Android 4.0源码文件较多，导入过程会持续较长一段时间，大概10几分钟左右。

注意 导入源码前一定要取消Eclipse的自动编译选项（通过菜单栏Project->Build Project Automatically设置）。另外，源码导入完毕后，读者千万不要清理（clean）这个工程。清理会删除之前源码编译所生成的文件，导致后续又得重新编译Android系统了。

4. 创建并运行模拟器

单击Eclipse菜单栏Window->AVD Manager，创建模拟器，如图1-13所示。

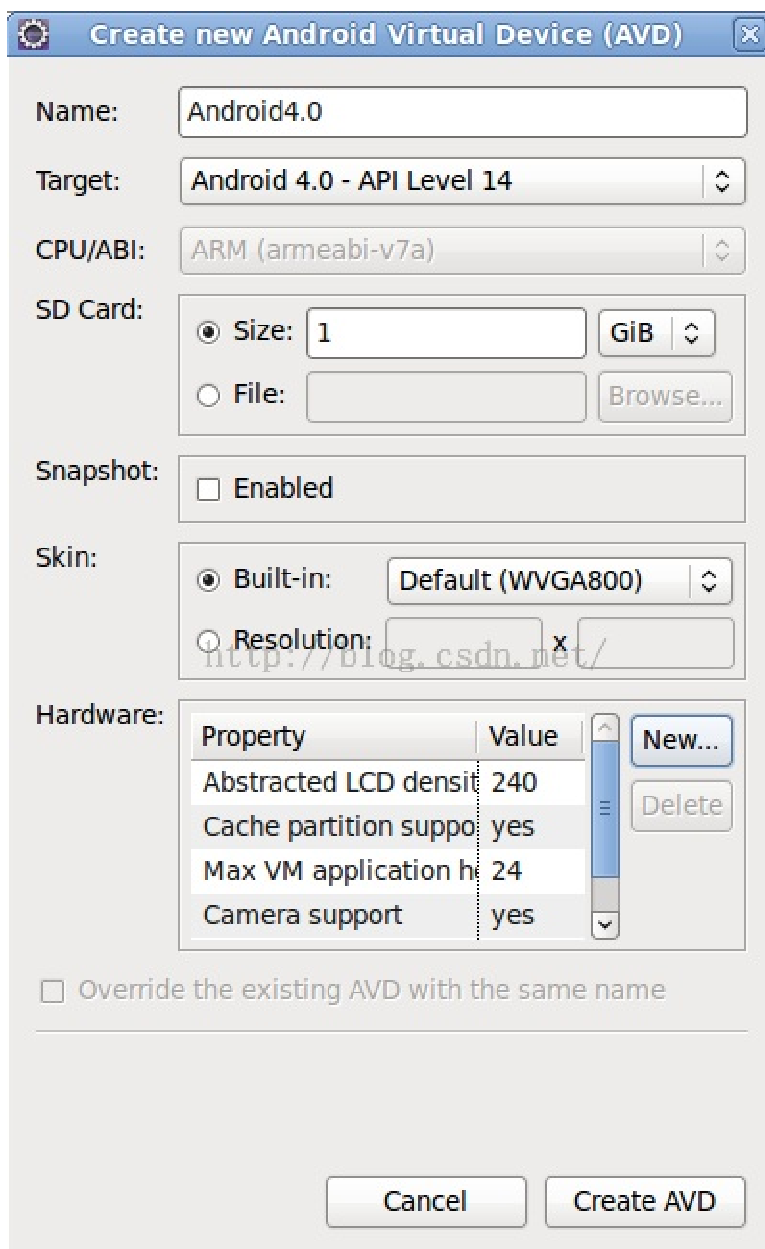


图1-13 模拟器创建示意图

模拟器创建完毕后即可启动它。

5. 调试SystemServer

调试SystemServer的步骤如下：

- 首先编译Android源码工程。编译过程中会有很多警告。如果有错误，大部分原因是.classpath文件将不需要的模块包含了进来。读者可根据Eclipse的提示做相应处理。笔者配置的几台机器基本都是一次配置就成功了。
- 在Android源码工程上单击右键，依次单击Debug As->Debug Configurations，弹出如图1-14所示的对话框，然后从左边找到RemoteJava Application一栏。

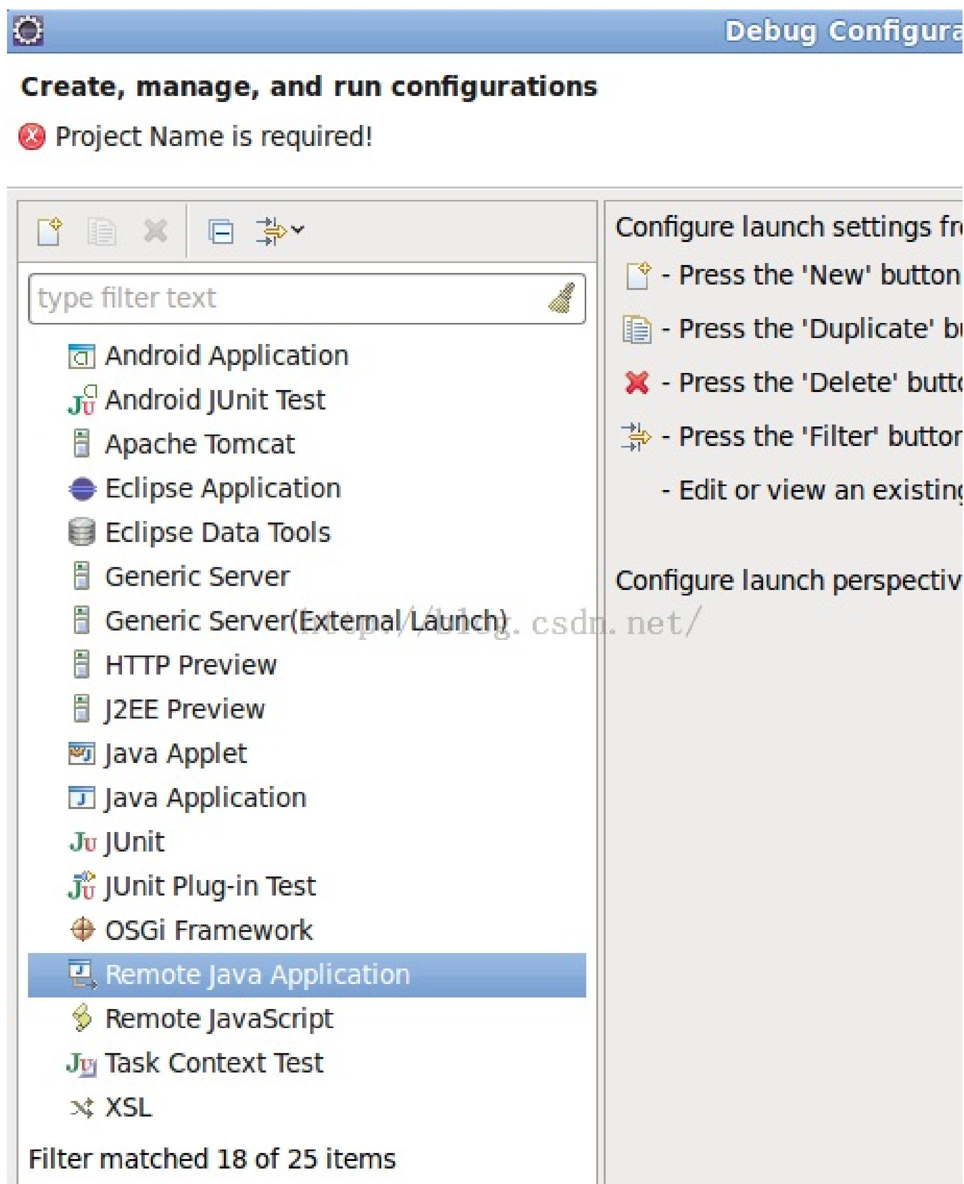


图1-14 Debug配置框示意图

- 单击图1-14中黑框中的新建按钮，然后按图1-15中的黑框中的内容来设置该对话框。

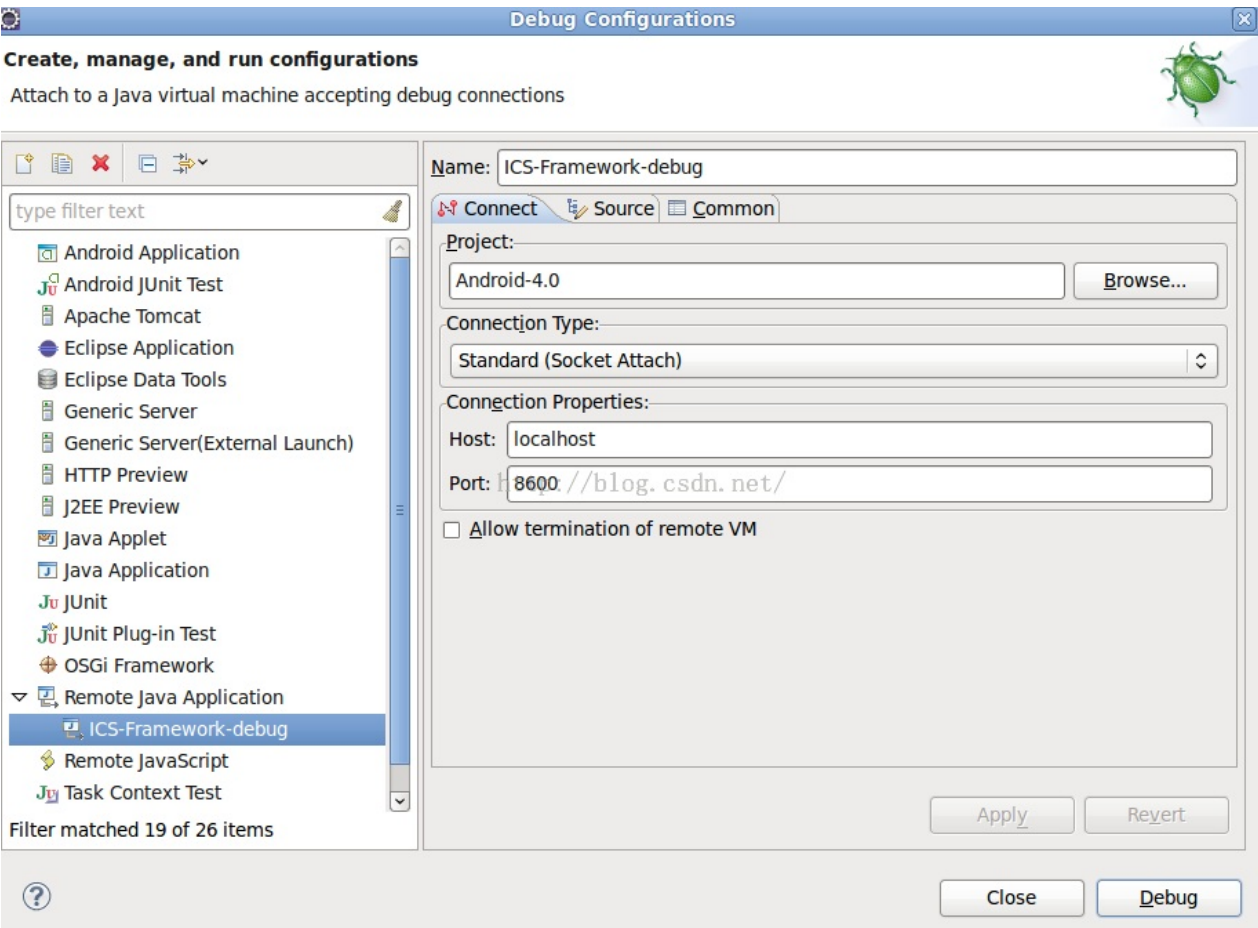


图1-15 Remote Java Application配置示意图

由图1-15所示，需要选择Remote调试端口号为8600，Host类型为localhost。8600是SystemServer进程的调试端口号。Eclipse一旦连接到该端口，即可通过JDWP协议来调试SystemServer。

- 配置完毕后，单击图1-15右下角的Debug按钮，即可启动SystemServer的调试。

图1-16所示为笔者调试startActivity流程的示意图。

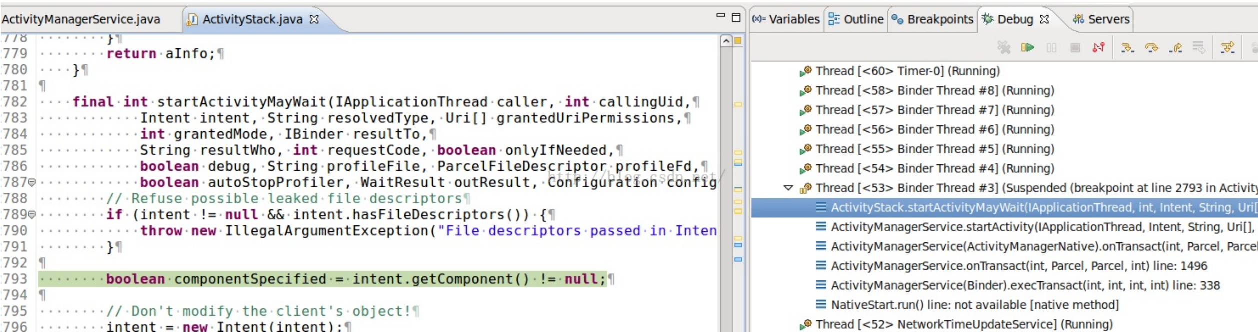


图1-16 SystemServer调试效果图

1.3 本章小结

本章对Android系统和源码搭建，以及如何利用Eclipse调试SystemServer等做了相关介绍，相信读者现在已经迫不及待了吧？马上开始我们的源码征程！

第2章 深入理解Java Binder和MessageQueue

本章主要内容：

- 介绍Binder系统的Java层框架
- 介绍MessageQueue

本章所涉及的源代码文件名及位置：

- IBinder.java

frameworks/base/core/java/android/os/IBinder.java

- Binder.java

frameworks/base/core/java/android/os/Binder.java

- BinderInternal.java

frameworks/base/core/java/com/android/intenal/os/BinderInternal.java

- android_util_Binder.cpp

frameworks/base/core/jni/android_util_Binder.cpp

- SystemServer.java

frameworks/base/services/java/com/android/servers/SystemServer.java

- ActivityManagerService.java

frameworks/base/services/java/com/android/servers/ActivityManagerService.java

- ServiceManager.java

frameworks/base/core/java/android/os/ServiceManager.java

- ServcieManagerNative.java

frameworks/base/core/java/android/os/ServcieManagerNative.java

- MessageQueue.java

frameworks/base/core/java/android/os/MessageQueue.java

- android_os_MessageQueue.cpp

frameworks/base/core/jni/android_os_MessageQueue.cpp

- Looper.cpp

frameworks/base/native/android/Looper.cpp

- Looper.h

frameworks/base/include/utils/Looper.h

- android_app_NativeActivity.cpp

frameworks/base/core/jni/android_app_NativeActivity.cpp

2.1 概述

由于本书所介绍的内容主要是以Java层的系统服务为主，因此Binder相关的应用在本书中比比皆是。而MessageQueue作为Android中重要的任务调度工具，它的使用也是随处可见。所以本书有必要对这两个工具有所介绍。根据邓凡平的同意与推荐，本章由卷II第2章升级到4.2.2而来，并且增加了对AIDL相关的知识点的分析。

以本章作为本书Android分析之旅的开篇，将重点关注两个基础知识点，它们是：

- Binder系统在Java世界是如何布局和工作的。
- MessageQueue的新职责。

先来分析Java层中的Binder。

建议读者先阅读《深入理解Android：卷I》（以下简称“卷I”）的第6章“深入理解Binder”。网上有样章可下载。

2.2 Java层中的Binder分析

2.2.1 Binder架构总览

如果读者读过卷I第6章“深入理解Binder”，相信就不会对Binder架构中代表Client的Bp端及代表Server的Bn端感到陌生。Java层中Binder实际上也是一个C/S架构，而且其在类的命名上尽量保持与Native层一致，因此可认为，Java层的Binder架构是Native层Binder架构的一个镜像。Java层的Binder架构中的成员如图2-1所示。

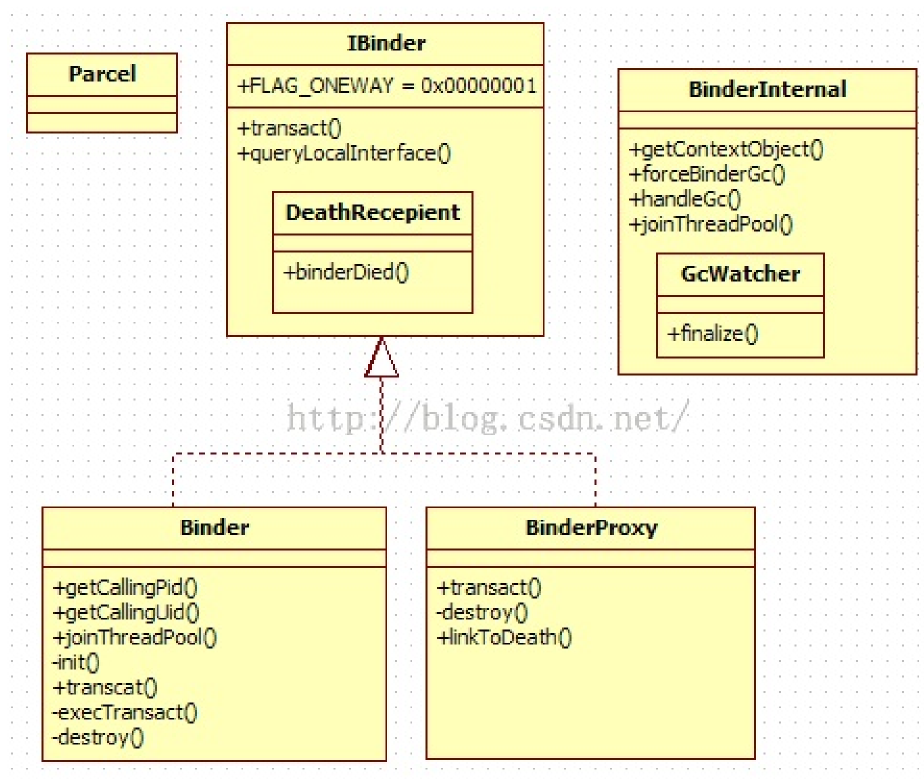


图 2 - 1 Java层中的Binder家族

由图2-1可知：

- 系统定义了一个IBinder接口类以及DeathRecepient接口。
- Binder类和BinderProxy类分别实现了IBinder接口。其中Binder类作为服务端的Bn的代表，而BinderProxy作为客户端的Bp的代表。
- 系统中还定义一个BinderInternal类。该类是一个仅供Binder框架使用的类。它内部有一个GcWatcher类，该类专门用于处理和Binder相关的垃圾回收。
- Java层同样提供一个用于承载通信数据的Parcel类。

注意 IBinder接口类中定义了一个叫FLAG_ONEWAY的整型，该变量的意义非常重要。当客户端利用Binder机制发起一个跨进程的函数调用时，调用方（即客户端）一般会阻塞，直到服务端返回结果。这种方式和普通的函数调用是一样的。但是在调用Binder函数时，在指明了FLAG_ONEWAY标志后，调用方只要把请求发送到Binder驱动即可返回，而不用等待服务端的结果，这就是一种所谓的非阻塞方式。在Native层中，涉及的Binder调用基本都是阻塞的，但是在Java层的framework中，使用FLAG_ONEWAY进行Binder调用的情况非常多，以后经常会碰到。

思考 使用FLAG_ONEWAY进行函数调用的程序在设计上有什么特点？这里简单分析一下：对于使用FLAG_ONEWAY的函数来说，客户端仅向服务端发出了请求，但是并不能确定服务端是否处理了该请求。所以，客户端一般会向服务端注册一个回调（同样是跨进程的Binder调用），一旦服务端处理了该请求，就会调用此回调来通知客户端处理结果。当然，这种回调函数也大多采用FLAG_ONEWAY的方式。

2.2.2 初始化Java层Binder框架

虽然Java层Binder系统是Native层Binder系统的一个Mirror，但这个Mirror终归还需借助Native层Binder系统来开展工作，即Mirror和Native层Binder有着千丝万缕的关系，一定要在Java层Binder正式工作之前建立这种关系。下面分析Java层Binder框架是如何初始化的。

在Android系统中，在Java初创时期，系统会提前注册一些JNI函数，其中有一个函数专门负责搭建Java Binder和Native Binder交互关系，该函数是register_android_os_Binder，代码如下：

```
[android_util_Binder.cpp-->register_android_os_Binder()]
int register_android_os_Binder(JNIEnv* env)
{
    // 初始化Java Binder类和Native层的关系
    if(int_register_android_os_Binder(env) &lt; 0)
       return -1;
    // 初始化Java BinderInternal类和Native层的关系
    if(int_register_android_os_BinderInternal(env) &lt; 0)
       return -1;
    // 初始化Java BinderProxy类和Native层的关系
    if(int_register_android_os_BinderProxy(env) &lt; 0)
       return -1;
    ......
    return0;
}
```

据上面的代码可知，register_android_os_Binder函数完成了Java Binder架构中最重要的3个类的初始化工作。

1. Binder类的初始化

int_register_android_os_Binder函数完成了Binder类的初始化工作，代码如下：

```
[android_util_Binder.cpp-->int_register_android_os_Binder()]
static int int_register_android_os_Binder(JNIEnv*env)
{
    jclassclazz;
   //kBinderPathName为Java层中Binder类的全路径名，“android/os/Binder“
    clazz =env->FindClass(kBinderPathName);
    /*gBinderOffSets是一个静态类对象，它专门保存Binder类的一些在JNI层中使用的信息，
      如成员函数execTranscat的methodID,Binder类中成员mObject的fildID */
   gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
   gBinderOffsets.mExecTransact
                    =env->GetMethodID(clazz, "execTransact", "(IIII)Z");
   gBinderOffsets.mObject
                    =env->GetFieldID(clazz, "mObject", "I");
    // 注册Binder类中native函数的实现
    returnAndroidRuntime::registerNativeMethods(
                            env,kBinderPathName,
                            gBinderMethods,NELEM(gBinderMethods));
}
```

从上面代码可知，gBinderOffsets对象保存了和Binder类相关的某些在JNI层中使用的信息。它们将用来在JNI层对Java层的Binder对象进行操作。execTransact()函数以及mObject成员的用途将在2.2.3节介绍。

建议 如果读者对JNI不是很清楚，可参阅卷I第2章“深入理解JNI”。

2. BinderInternal类的初始化

下一个初始化的类是BinderInternal，其代码在int_register_android_os_BinderInternal函数中。

```
[android_util_Binder.cpp-->int_register_android_os_BinderInternal()]
static intint_register_android_os_BinderInternal(JNIEnv* env)
{
   jclassclazz;
   // 根据BinderInternal的全路径名找到代表该类的jclass对象。全路径名为
   //“com/android/internal/os/BinderInternal”
   clazz =env->FindClass(kBinderInternalPathName);
  //gBinderInternalOffsets也是一个静态对象，用来保存BinderInternal类的一些信息
  gBinderInternalOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
   // 获取forceBinderGc的methodID
  gBinderInternalOffsets.mForceGc
               = env->GetStaticMethodID(clazz, "forceBinderGc","()V");
   // 注册BinderInternal类中native函数的实现
   returnAndroidRuntime::registerNativeMethods(
                         env,kBinderInternalPathName,
                        gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

int_register_android_os_BinderInternal的工作内容和int_register_android_os_Binder的工作内容类似：

- 获取一些有用的methodID和fieldID。这表明JNI层一定会向上调用Java层的函数。
- 注册相关类中native函数的实现。

3. BinderProxy类的初始化

int_register_android_os_BinderProxy完成了BinderProxy类的初始化工作，代码稍显复杂，如下所示：

```
[android_util_Binder.cpp-->int_register_android_os_BinderProxy()]
static intint_register_android_os_BinderProxy(JNIEnv* env)
{
    jclassclazz;
    // **① gWeakReferenceOffsets用来和WeakReference类打交道**
    clazz =env->FindClass("java/lang/ref/WeakReference");
   gWeakReferenceOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
    // 获取WeakReference类get函数的MethodID
   gWeakReferenceOffsets.mGet= env->GetMethodID(clazz, "get",
                                "()Ljava/lang/Object;");
    // **② gErrorOffsets用来和Error类打交道**
    clazz =env->FindClass("java/lang/Error");
   gErrorOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
    // **③ gBinderProxyOffsets用来和BinderProxy类打交道**
    clazz =env->FindClass(kBinderProxyPathName);
   gBinderProxyOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
   gBinderProxyOffsets.mConstructor= env->GetMethodID(clazz,"&lt;init&gt;", "()V");
    ......//获取BinderProxy的一些信息
    // **④ gClassOffsets用来和Class类打交道**
    clazz =env->FindClass("java/lang/Class");
   gClassOffsets.mGetName =env->GetMethodID(clazz,
                           "getName","()Ljava/lang/String;");
    // 注册BinderProxy native函数的实现
    returnAndroidRuntime::registerNativeMethods(env,
         kBinderProxyPathName,gBinderProxyMethods,
                            NELEM(gBinderProxyMethods));
}
```

据上面代码可知，int_register_android_os_BinderProxy函数除了初始化BinderProxy类外，还获取了WeakReference类和Error类的一些信息。看来BinderProxy对象的生命周期会委托WeakReference来管理，难怪JNI层会获取该类get函数的MethodID。

至此，Java Binder几个重要成员的初始化已完成，同时在代码中定义了几个全局静态对象，分别是gBinderOffsets、gBinderInternalOffsets和gBinderProxyOffsets。

框架的初始化其实就是提前获取一些JNI层的使用信息，如类成员函数的MethodID，类成员变量的fieldID等。这项工作是必需的，因为它能节省每次使用时获取这些信息的时间。当Binder调用频繁时，这些时间累积起来还是不容小觑的。

另外，这个过程中所创建的几个全局静态对象为JNI层访问Java层的对象提供了依据。而在每个初始化函数中所执行的registerNativeMethods()方法则为Java层访问JNI层打通了道路。换句话说，Binder初始化的工作就是通过JNI建立起Native Binder与Java Binder之间互相通信的桥梁。

下面通过一个例子来分析Java Binder的工作流程。

2.2.3 窥一斑，可见全豹乎

这个例子源自ActivityManagerService，我们试图通过它揭示Java层Binder的工作原理。先来描述一下该例子的分析步骤：

- 首先分析AMS如何将自己注册到ServiceManager。
- 然后分析AMS如何响应客户端的Binder调用请求。

本例的起点是setSystemProcess，其代码如下所示：

```
[ActivityManagerService.java-->ActivityManagerService.setSystemProcess()]
public static void setSystemProcess() {
   try {
      ActivityManagerService m = mSelf;
       // 将ActivityManagerService服务注册到ServiceManager中
      ServiceManager.addService("activity", m);......
   } catch {... }
   return;
}
```

上面所示代码行的目的是将ActivityManagerService服务（以后简称AMS）加到ServiceManager中。

在整个Android系统中有一个Native的ServiceManager（以后简称SM）进程，它统筹管理Android系统上的所有Service。成为一个Service的首要条件是先在SM中注册。下面来看Java层的Service是如何向SM注册的。

1. 向ServiceManager注册服务

（1）创建ServiceManagerProxy

向SM注册服务的函数叫addService，其代码如下：

```
[ServiceManager.java-->ServiceManager.addService()]
public static void addService(String name, IBinderservice) {
    try {
        //getIServiceManager返回什么
       getIServiceManager().addService(name, service);
    }
    ......
}
```

首先需要搞清楚getIServiceManager()方法返回的是一个什么对象呢？参考其实现：

```
[ServiceManager.java-->ServiceManager.getIServiceManager()]
private static IServiceManagergetIServiceManager() {
    ......
    // 调用asInterface，传递的参数类型为IBinder
   sServiceManager = ServiceManagerNative.asInterface(
                       BinderInternal.getContextObject());
    returnsServiceManager;
}
```

asInterface()方法的参数为BinderInternal.getContextObject()的返回值。于是这个简短的方法中有两个内容值得讨论：BinderInternal.getContextObject()以及asInterface()。

BinderInternal.getContextObject()方法是一个native的函数，参考其实现：

```
[android_util_Binder.cpp-->android_os_BinderInternal_getContextObject()]
static jobjectandroid_os_BinderInternal_getContextObject(JNIEnv* env, jobject clazz)
{
    /* 下面这句代码在卷I第6章详细分析过，它将返回一个BpProxy对象，其中
      NULL（即0，用于标识目的端）指定Proxy通信的目的端是ServiceManager*/
   sp&lt;IBinder&gt; b = ProcessState::self()->getContextObject(NULL);
    // 由Native对象创建一个Java对象,下面分析该函数
    returnjavaObjectForIBinder(env, b);
}
```

可见，Java层的ServiceManager需要在Native层获取指向Native进程中ServiceManager的BpProxy。这个BpProxy不能由Java层的ServiceManager直接使用，于是android_os_BinderInteral_getContextObject()函数通过javaObjectForIBinder()函数将创建一个封装了这个BpProxy的一个Java对象并返回给调用者。ServiceManager便可一通过这个Java对象实现对BpProxy的访问。参考这个Java对象的创建过程：

```
[android_util_Binder.cpp-->javaObjectForIBinder()]
jobject javaObjectForIBinder(JNIEnv* env, constsp&lt;IBinder&gt;& val)
{
    //mProxyLock是一个全局的静态CMutex对象
   AutoMutex _l(mProxyLock);
    /* val对象实际类型是BpBinder，读者可自行分析BpBinder.cpp中的findObject函数。
      事实上，在Native层的BpBinder中有一个ObjectManager，它用来管理在Native BpBinder
      上创建的Java BpBinder对象。下面这个findObject用来判断gBinderProxyOffsets
      是否已经保存在ObjectManager中。如果是，那就需要删除这个旧的object */
    jobjectobject = (jobject)val->findObject(&gBinderProxyOffsets);
    if(object != NULL) {
       jobject res = env->CallObjectMethod(object,gWeakReferenceOffsets.mGet);
       android_atomic_dec(&gNumProxyRefs);
       val->detachObject(&gBinderProxyOffsets);
       env->DeleteGlobalRef(object);
    }
    // **① 创建一个新的BinderProxy对象。**并将它注册到Native BpBinder对象的ObjectManager中
    object =env->NewObject(gBinderProxyOffsets.mClass,
                           gBinderProxyOffsets.mConstructor);
    if(object != NULL) {
        /* ② 把Native层的BpProxy的指针保存到BinderProxy对象的成员字段mObject中。
          于是BinderProxy对象的Native方法可以通过mObject获取BpProxy对象的指针。
          这个操作是将BinderProxy与BpProxy联系起来的纽带 */
       env->SetIntField(object, gBinderProxyOffsets.mObject,(int)val.get());
       val->incStrong(object);
       jobject refObject = env->NewGlobalRef(
               env->GetObjectField(object, gBinderProxyOffsets.mSelf));
        /* 将这个新创建的BinderProxy对象注册（attach）到BpBinder的ObjectManager中，
          同时注册一个回收函数proxy_cleanup。当BinderProxy对象撤销（detach）的时候，
          该函数会被调用，以释放一些资源。读者可自行研究proxy_cleanup函数*/
       val->attachObject(&gBinderProxyOffsets, refObject,
                              jnienv_to_javavm(env), proxy_cleanup);
        //DeathRecipientList保存了一个用于死亡通知的list
       sp&lt;DeathRecipientList&gt; drl = new DeathRecipientList;
       drl->incStrong((void*)javaObjectForIBinder);
        //将死亡通知list和BinderProxy对象联系起来
       env->SetIntField(object, gBinderProxyOffsets.mOrgue,
                             reinterpret_cast&lt;jint&gt;(drl.get()));
        // 增加该Proxy对象的引用计数
       android_atomic_inc(&gNumProxyRefs);
        /*下面这个函数用于垃圾回收。创建的Proxy对象一旦超过200个，该函数
          将调用BinderInter类的ForceGc做一次垃圾回收 */
       incRefsCreated(env);
    }
    returnobject;
}
```

BinderInternal.getContextObject的代码有点多，简单整理一下，可知该函数完成了以下两个工作：

- 创建了一个Java层的BinderProxy对象。
- 通过JNI，该BinderProxy对象和一个Native的BpProxy对象挂钩，而该BpProxy对象的通信目标就是ServiceManager。

接下来讨论asInterface()方法，大家还记得在Native层Binder中那个著名的interface_cast宏吗？在Java层中，虽然没有这样的宏，但是定义了一个类似的函数asInterface。下面来分析ServiceManagerNative类的asInterface函数，其代码如下：

```
[ServiceManagerNative.java-->ServiceManagerNative.asInterface()]
static public IServiceManager asInterface(IBinderobj)
{
    ......// 以obj为参数，创建一个ServiceManagerProxy对象
    returnnew ServiceManagerProxy(obj);
}
```

上面代码和Native层interface_cast非常类似，都是以一个BpProxy对象为参数构造一个和业务相关的Proxy对象，例如这里的ServiceManagerProxy对象。ServiceManagerProxy对象的各个业务函数会将相应请求打包后交给BpProxy对象，最终由BpProxy对象发送给Binder驱动以完成一次通信。

说明 实际上BpProxy也不会直接和Binder驱动交互，真正和Binder驱动交互的是IPCThreadState。

（**2**）**addService**函数分析

现在来分析ServiceManagerProxy的addService函数，其代码如下：

```
[ServcieManagerNative.java-->ServiceManagerProxy.addService()]
public void addService(String name, IBinderservice)
                           throwsRemoteException {
    Parceldata = Parcel.obtain();
    Parcelreply = Parcel.obtain();
   data.writeInterfaceToken(IServiceManager.descriptor);
   data.writeString(name);
    // 注意下面这个writeStrongBinder函数，后面我们会详细分析它
   data.writeStrongBinder(service);
    /*mRemote实际上就是BinderProxy对象，调用它的transact，将封装好的请求数据
      发送出去 *
   mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);
    reply.recycle();
   data.recycle();
}
```

BinderProxy的transact，是一个native函数，其实现函数的代码如下所示：

```
[android_util_Binder.cpp-->android_os_BinderProxy_transact()]
static jbooleanandroid_os_BinderProxy_transact(JNIEnv* env, jobject obj,
                                           jint code, jobject dataObj,
                                           jobject replyObj, jint flags)
{
    ......
    // 从Java的Parcel对象中得到作为参数的Native的Parcel对象
    Parcel*data = parcelForJavaObject(env, dataObj);
    if (data== NULL) {
       return JNI_FALSE;
    }
    // 得到一个用于接收回复的Parcel对象
    Parcel*reply = parcelForJavaObject(env, replyObj);
    if(reply == NULL && replyObj != NULL) {
       return JNI_FALSE;
    }
    // 从Java的BinderProxy对象中得到之前已经创建好的那个Native的BpBinder对象
    IBinder*target = (IBinder*)
       env->GetIntField(obj, gBinderProxyOffsets.mObject);
    ......
    // 通过Native的BpBinder对象，将请求发送给ServiceManager
    status_terr = target->transact(code, *data, reply, flags);
    ......
    signalExceptionForError(env,obj, err);
    returnJNI_FALSE;
}
```

看了上面的代码会发现，Java层的Binder最终还是要借助Native的Binder进行通信的。

说明 从架构的角度看，在Java中搭建了一整套框架，如IBinder接口，Binder类和BinderProxy类。但是从通信角度看，不论架构的编写采用的是Native语言还是Java语言，只要把请求传递到Binder驱动就可以了，所以通信的目的是向binder发送请求和接收回复。在这个目的之上，考虑到软件的灵活性和可扩展性，于是编写了一个架构。反过来说，也可以不使用架构（即没有使用任何接口、派生之类的东西）而直接和binder交互，例如ServiceManager作为Binder的一个核心程序，就是直接读取/dev/binder设备，获取并处理请求。从这一点上看，Binder的目的虽是简单的（即打开binder设备，然后读请求和写回复），但是架构是复杂的（编写各种接口类和封装类等）。我们在研究源码时，一定要先搞清楚目的。实现只不过是达到该目的的一种手段和方式。脱离目的的实现，如缘木求鱼，很容易偏离事物本质。

在对addService进行分析时曾提示writeStrongBinder是一个特别的函数。那么它特别在哪里呢？下面将给出解释。

（3）三人行之**Binder、JavaBBinderHolder**和**JavaBBinder**

ActivityManagerService从ActivityManagerNative类派生，并实现了一些接口，其中和Binder的相关的只有这个ActivityManagerNative类，其原型如下：

```
[ActivityManagerNative.java-->ActivityManagerNative]
public abstract class ActivityManagerNative
                          extends Binder
                          implementsIActivityManager
```

ActivityManagerNative从Binder派生，并实现了IActivityManager接口。下面来看ActivityManagerNative的构造函数：

```
[ActivityManagerNative.java-->ActivityManagerNative.ActivityManagerNative()]
public ActivityManagerNative() {
   attachInterface(this, descriptor);// 该函数很简单，读者可自行分析
}
```

而ActivityManagerNative父类的构造函数则是Binder的构造函数：

```
[Binder.java-->Binder.Binder()]
public Binder() {
    init();
}
```

Binder构造函数中会调用native的init函数，其实现的代码如下：

```
[android_util_Binder.cpp-->android_os_Binder_init()]
static void android_os_Binder_init(JNIEnv* env,jobject obj)
{
    // 创建一个JavaBBinderHolder对象
   JavaBBinderHolder* jbh = new JavaBBinderHolder();
   bh->incStrong((void*)android_os_Binder_init);
    // 将这个JavaBBinderHolder对象保存到Java Binder对象的mObject成员中
   env->SetIntField(obj, gBinderOffsets.mObject, (int)jbh);
}
```

从上面代码可知，Java的Binder对象将和一个Native的JavaBBinderHolder对象相关联。那么，JavaBBinderHolder是何方神圣呢？其定义如下：

```
[android_util_Binder.cpp-->JavaBBinderHolder]
class JavaBBinderHolder : public RefBase
{
public:
   sp&lt;JavaBBinder&gt; get(JNIEnv* env, jobject obj)
    {
       AutoMutex _l(mLock);
       sp&lt;JavaBBinder&gt; b = mBinder.promote();
        if(b == NULL) {
           // 创建一个JavaBBinder，obj实际上是Java层中的Binder对象
           b = new JavaBBinder(env, obj);
           mBinder = b;
        }
       return b;
    }
    ......
private:
   Mutex           mLock;
   wp&lt;JavaBBinder&gt; mBinder;
};
```

从派生关系上可以发现，JavaBBinderHolder仅从RefBase派生，所以它不属于Binder家族。Java层的Binder对象为什么会和Native层的一个与Binder家族无关的对象绑定呢？仔细观察JavaBBinderHolder的定义可知：JavaBBinderHolder类的get函数中创建了一个JavaBBinder对象，这个对象就是从BnBinder派生的。

那么，这个get函数是在哪里调用的？答案在下面这句代码中：

```
//其中，data是Parcel对象，service此时还是ActivityManagerService
data.writeStrongBinder(service);
```

writeStrongBinder会做一个替换工作，下面是它的native代码实现：

```
[android_util_Binder.cpp-->android_os_Parcel_writeStrongBinder()]
static voidandroid_os_Parcel_writeStrongBinder(JNIEnv* env,
                                               jobject clazz, jobject object)
{
    /*parcel是一个Native的对象，writeStrongBinder的真正参数是
     ibinderForJavaObject()的返回值*/
    conststatus_t err = parcel->writeStrongBinder(
                               ibinderForJavaObject(env, object));
}
[android_util_Binder.cpp-->ibinderForJavaObject()]
sp&lt;IBinder&gt; ibinderForJavaObject(JNIEnv*env, jobject obj)
{
    /* 如果Java的obj是Binder类，则首先获得JavaBBinderHolder对象，然后调用
      它的get()函数。而这个get将返回一个JavaBBinder  */
    if(env->IsInstanceOf(obj, gBinderOffsets.mClass)) {
    JavaBBinderHolder* jbh = (JavaBBinderHolder*)env->GetIntField(obj,
                                 gBinderOffsets.mObject);
       return jbh != NULL ? jbh->get(env, obj) : NULL;
    }
    // 如果obj是BinderProxy类，则返回Native的BpBinder对象
    if(env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {
       return (IBinder*)
           env->GetIntField(obj, gBinderProxyOffsets.mObject);
    }
   returnNULL;
}
```

根据上面的介绍会发现，addService实际添加到Parcel的并不是AMS本身，而是一个叫JavaBBinder的对象。正是将它最终传递到Binder驱动。

读者此时容易想到，Java层中所有的Binder对应的都是这个JavaBBinder。当然，不同的Binder对象对应不同的JavaBBinder对象。

图2-2展示了Java Binder、JavaBBinderHolder和JavaBBinder的关系。

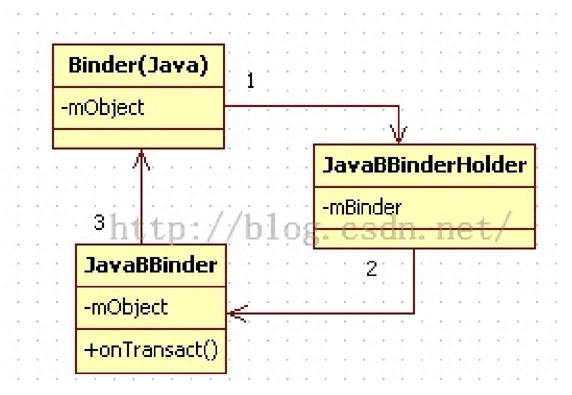


图 2 - 2 Java Binder 、JavaBBinderHolder和JavaBBinder三者的关系

从图2-2可知：

- Java层的Binder通过mObject指向一个Native层的JavaBBInderHolder对象。

- Native层的JavaBBinderHolder对象通过mBinder成员变量指向一个Native的JavaBBinder对象。
- Native的JavaBBinder对象又通过mObject变量指向一个Java层的Binder对象。

为什么不直接让Java层的Binder对象指向Native层的JavaBBinder对象呢？由于缺乏设计文档，这里不便妄加揣测，但从JavaBBinderHolder的实现上来分析，估计和垃圾回收（内存管理）有关，因为JavaBBinderHolder中的mBinder对象的类型被定义成弱引用wp了。

建议 对此有更好的解释的读者，不妨与大家分享一下。

2. ActivityManagerService响应请求

初见JavaBBinde时，多少有些吃惊。回想一下Native层的Binder架构：虽然在代码中调用的是Binder类提供的接口，但其对象却是一个实际的服务端对象，例如MediaPlayerService对象，AudioFlinger对象。

而在Java层的Binder架构中，JavaBBinder却是一个和业务完全无关的对象。那么，这个对象如何实现不同业务呢？

为回答此问题，我们必须看它的onTransact函数。当收到请求时，系统会调用这个函数。

说明 关于这个问题，建议读者阅读卷I第6章“深入理解Binder”。

```
[android_util_Binder.cpp-->JavaBBinder::onTransact()]
virtual status_t onTransact(
       uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags =0)
{
    JNIEnv*env = javavm_to_jnienv(mVM);
   IPCThreadState* thread_state = IPCThreadState::self();
    .......
    // 调用Java层Binder对象的execTranscat函数
    jbooleanres = env->CallBooleanMethod(mObject,
                    gBinderOffsets.mExecTransact,code,
                   (int32_t)&data, (int32_t)reply, flags);
    ......
    returnres != JNI_FALSE ? NO_ERROR : UNKNOWN_TRANSACTION;
}
```

就本例而言，上面代码中的mObject就是ActivityManagerService，现在调用它的execTransact()方法，该方法在Binder类中实现，具体代码如下：

```
[Binder.java-->Binder.execTransact()]
private boolean execTransact(int code, intdataObj, int replyObj,int flags) {
    Parceldata = Parcel.obtain(dataObj);
    Parcelreply = Parcel.obtain(replyObj);
    booleanres;
    try {
        //调用onTransact函数，派生类可以重新实现这个函数，以完成业务功能
        res= onTransact(code, data, reply, flags);
    } catch{ ... }
   reply.recycle();
   data.recycle();
    returnres;
}
```

ActivityManagerNative类实现了onTransact函数，代码如下：

```
[ActivityManagerNative.java-->ActivityManagerNative.onTransact()]
public boolean onTransact(int code, Parcel data,Parcel reply, int flags)
           throws RemoteException {
    switch(code) {
    caseSTART_ACTIVITY_TRANSACTION:
    {
       data.enforceInterface(IActivityManager.descriptor);
       IBinder b = data.readStrongBinder();
       ......
        //再由ActivityManagerService实现业务函数startActivity
        intresult = startActivity(app, intent, resolvedType,
                   grantedUriPermissions, grantedMode, resultTo, resultWho,
                   requestCode, onlyIfNeeded, debug, profileFile,
                   profileFd, autoStopProfiler);
       reply.writeNoException();
       reply.writeInt(result);
       return true;
    }
    .... // 处理其他请求的case
    }
}
```

由此可以看出，JavaBBinder仅是一个传声筒，它本身不实现任何业务函数，其工作是：

- 当它收到请求时，只是简单地调用它所绑定的Java层Binder对象的exeTransact。
- 该Binder对象的exeTransact调用其子类实现的onTransact函数。
- 子类的onTransact函数将业务又派发给其子类来完成。请读者务必注意其中的多层继承关系。

通过这种方式，来自客户端的请求就能传递到正确的Java Binder对象了。图2-3展示AMS响应请求的整个流程。

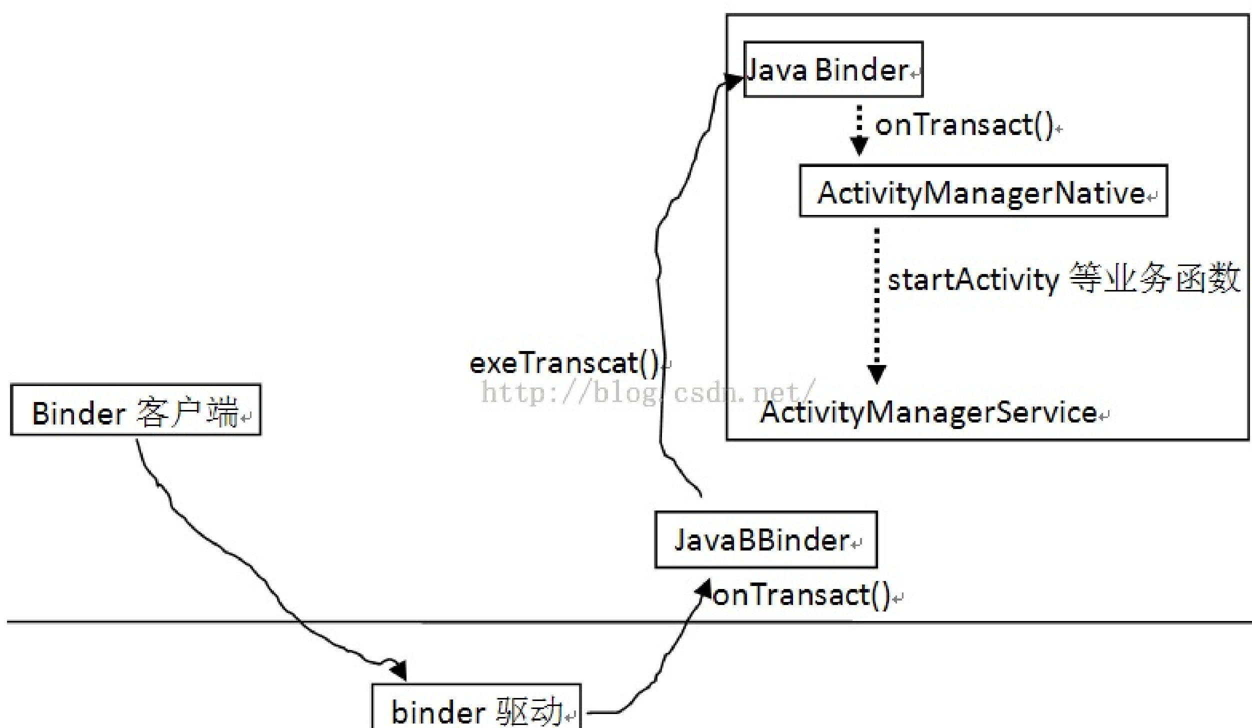


图 2 - 3 AMS响应请求的流程

在图2-3中，右上角的大方框表示AMS这个对象，其间的虚线箭头表示调用子类重载的函数。

2.2.4 理解AIDL

经过上一节的介绍，读者已经明白在Java层Binder的架构中，Bp端可以通过BinderProxy的transact()方法与Bn端发送请求，而Bn端通过继承Binder类并重写onTransact()接收并处理来自Bp端的请求。这个结构非常清晰而且简单，但是实现起来却颇为繁琐。于是Android提供了AIDL语言以及AIDL解释器自动生成一个服务的Bn端即Bp端的用于处理Binder通信的代码。

AIDL的语法与定义一个Java接口的语法非常相似。为了避免业务实现对分析的干扰，本节通过一个最简单的例子对AIDL的原理进行介绍：

```
[IMyServer.aidl]
package com.understanding.samples;
interface IMyServer {
    intfoo(String str);
}
```

IMyServer.aidl定义了一个名为IMyServer的Binder服务，并提供了一个可以跨Binder调用的接口foo()。可以通过aidl工具将其解析为一个实现了Bn端及Bp端通过Binder进行通信的Java源代码。具体命令如下：

```
aidl com/understanding/samples/IMyServer.aidl
```

生成的IMyServer.java可以在com/understanding/samples/文件夹下找到。

建议 读者可以阅读aidl有关的文档了解此工具的详细功能。

```
[IMyServer.java-->IMyServer]
package com.understanding.samples;
/* **① 首先，IMyServer.aidl被解析为一个Java接口IMyServer。**这个接口定义了AIDL文件中
  所定义的接口foo() */
public interface IMyServer extendsandroid.os.IInterface {
    /***② aidl工具生成了一个继承自IMyServer接口的抽象类IMyServer.Stub。**这个抽象类实现了
      Bn端通过onTransact()方法接收来自Bp端的请求的代码。本例中的foo()方法在这个类中
      会被定义成一个抽象方法。因为aidl工具根不知道foo()方法是做什么的，它只能在onTransact()
      中得知Bp端希望对foo()方法进行调用，所以Stub类是抽象的。  */
    publicstatic abstract class Stub extends android.os.Binder implements
                                       com.understanding.samples.IMyServer{
       ...... // Stub类的其他实现
        /*onTransact()根据code的值选择调用IMyServer接口中的不同方法。本例中
         TRANSACTION_foo意味着需要通过调用foo()方法完成请求 */
       public boolean onTransact(int code, android.os.Parcel data,
               android.os.Parcel reply, int flags)
               throws android.os.RemoteException {
           switch (code) {
            ......
           case TRANSACTION_foo: {
               ...... // 从data中读取参数_arg0
               // Stub类的子类需要实现foo()方法
                int _result = this.foo(_arg0);
               ...... // 向reply中写入_result
               return true;
           }
           }
           return super.onTransact(code, data, reply, flags);
        }
        /* **③ aidl工具还生成了一个继承自IMyServer接口的类Proxy，它是Bp端的实现。**与Bn端的
         Stub类不同，它实现了foo()函数。因为foo()函数在Bp端的实现是确定的，即将参数存储到
         Parcel中然后执行transact()方法将请求发送给Bn端，然后从reply中读取返回值并返回
          给调用者 */
          private static class Proxy implements com.understanding.samples.IMyServer{
            ...... // Proxy类的其他实现
           public int foo(java.lang.String str)
                   throws android.os.RemoteException {
               android.os.Parcel _data = android.os.Parcel.obtain();
               android.os.Parcel _reply = android.os.Parcel.obtain();
               int _result;
               try {
                   ...... // 将参数str写入参数_data
                   // mRemote就是指向IMyServer Bn端的BinderProxy
                   mRemote.transact(Stub.TRANSACTION_foo, _data, _reply, 0);
                   ......// 从_replay中读取返回值_result
               } finally { ...... }
               return _result;
           }
        }
        // TRANSACTION_foo常量用于定义foo()方法的code
          static final int TRANSACTION_foo =
                             (android.os.IBinder.FIRST_CALL_TRANSACTION+ 0);
    }
    // 声明IMyServer所提供的接口
    publicint foo(java.lang.String str) throws android.os.RemoteException;
}
```

可见一个AIDL文件被aidl工具解析之后会产生三个物件：

- IMyServer接口。它仅仅用来在Java中声明IMyServer.aidl中所声明的接口。
- IMyServer.Stub类。这个继承自Binder类的抽象类实现了Bn端与Binder通信相关的代码。
- IMyServer.Stub.Proxy类。这个类实现了Bp端与Binder通信相关的代码。

在完成了aidl的解析之后，为了实现一个Bn端，开发者需要继承IMyServer.Stub类并实现其抽象方法。如下所示：

```
class MyServer extends IMyServer.Stub {
    intfoo(String str) {
        // 做点什么都可以
        returnstr.length();
    }
}
```

于是每一个MyServer类的实例，都具有了作为Bn端的能力。典型的做法是将MyServer类的实例通过ServiceManager.addService()将其注册为一个系统服务，或者在一个Android标准Service的onBind()方法中将其作为返回值使之可以被其他进程访问。另外，也可以通过Binder调用将其传递给另外一个进程，使之成为一个跨进程的回调对象。

那么Bp端将如何使用IMyServer.Proxy呢？在Bp端所在进程中，一旦获取了IMyServer的BinderProxy（通过ServiceManager.getService()、onServiceConnected()或者其他方式），就可以以如下方式获得一个IMyServer.Proxy：

```
// 其中binderProxy就是通过ServiceManager.getService()所获取
IMyServer remote = IMyServer.Stub.asInterface(binderProxy);
remote.foo(“Hello AIDL!”);
IMyServer.Stub.asInterface()的实现如下：
[IMyServer.java-->IMyServer.Stub.asInterface()]
public static com.understanding.samples.IMyServerasInterface(
       android.os.IBinder obj) {
    ......
    // 创建一个IMyServer.Stub.Proxy。其中参数obj将会被保存为Proxy类的mRemote成员。
    return new com.understanding.samples.IMyServer.Stub.Proxy(obj);
}
```

可见，AIDL使得构建一个Binder服务的工作大大地简化了。

2.2.5 Java层Binder架构总结

图2-4展示了Java层的Binder架构。

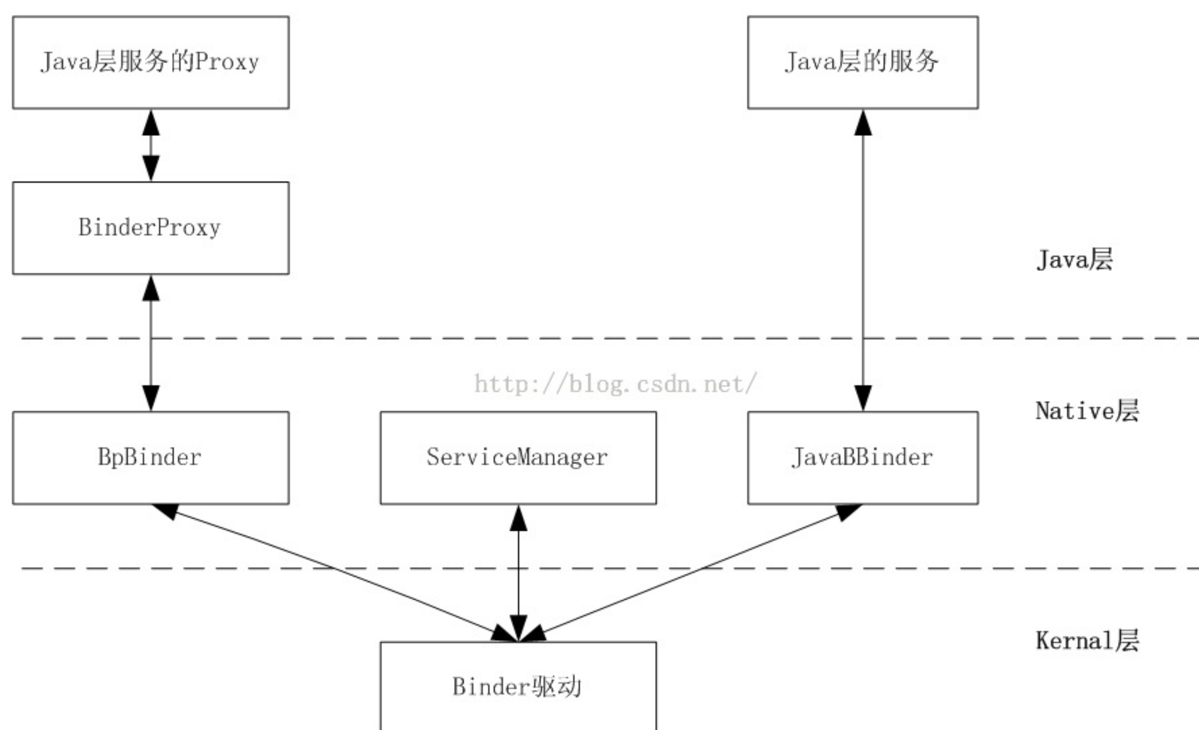


图 2 - 4 Java层Binder架构

根据图2-4可知：

- 对于代表客户端的BinderProxy来说，Java层的BinderProxy在Native层对应一个BpBinder对象。凡是从Java层发出的请求，首先从Java层的BinderProxy传递到Native层的BpBinder，继而由BpBinder将请求发送到Binder驱动。
- 对于代表服务端的Service来说，Java层的Binder在Native层有一个JavaBBinder对象。前面介绍过，所有Java层的Binder在Native层都对应为JavaBBinder，而JavaBBinder仅起到中转作用，即把来自客户端的请求从Native层传递到Java层。
- 系统中依然只有一个Native的ServiceManager。

至此，Java层的Binder架构已介绍完毕。从前面的分析可以看出，Java层Binder非常依赖Native层的Binder。建议想进一步了解Binder的读者们，要深入了解这一问题，有必要阅读卷I的第6章“深入理解Binder”。

2.3 心系两界的MessageQueue

卷I第5章介绍过，MessageQueue类封装了与消息队列有关的操作。在一个以消息驱动的系统中，最重要的两部分就是消息队列和消息处理循环。在Andrid 2.3以前，只有Java世界的居民有资格向MessageQueue中添加消息以驱动Java世界的正常运转，但从Android 2.3开始，MessageQueue的核心部分下移至Native层，让Native世界的居民也能利用消息循环来处理他们所在世界的事情。因此现在的MessageQueue心系Native和Java两个世界。

2.3.1 MessageQueue的创建

现在来分析MessageQueue是如何跨界工作的，其代码如下：

```
[MessageQueue.java-->MessageQueue.MessageQueue()]
MessageQueue() {
    nativeInit();//构造函数调用nativeInit，该函数由Native层实现
}
```

nativeInit()方法的真正实现为android_os_MessageQueue_nativeInit()函数，其代码如下：

```
[android_os_MessageQueue.cpp-->android_os_MessageQueue_nativeInit()]
static voidandroid_os_MessageQueue_nativeInit(JNIEnv* env, jobject obj) {
    // NativeMessageQueue是MessageQueue在Native层的代表
    NativeMessageQueue* nativeMessageQueue = newNativeMessageQueue();
    ......
    // 将这个NativeMessageQueue对象设置到Java层保存
    android_os_MessageQueue_setNativeMessageQueue(env,obj,
                                                    nativeMessageQueue);
}
```

nativeInit函数在Native层创建了一个与MessageQueue对应的NativeMessageQueue对象，其构造函数如下：

```
[android_os_MessageQueue.cpp-->NativeMessageQueue::NativeMessageQueue()]
NativeMessageQueue::NativeMessageQueue() {
    /* 代表消息循环的Looper也在Native层中呈现身影了。根据消息驱动的知识，一个线程会有一个
      Looper来循环处理消息队列中的消息。下面一行的调用就是取得保存在线程本地存储空间
      (Thread Local Storage）中的Looper对象 */
    mLooper= Looper::getForThread();
    if (mLooper == NULL) {
        /* 如为第一次进来，则该线程没有设置本地存储，所以须先创建一个Looper，然后再将其保存到
          TLS中，这是很常见的一种以线程为单位的单例模式*/
        mLooper = new Looper(false);
        Looper::setForThread(mLooper);
    }
}
```

Native的Looper是Native世界中参与消息循环的一位重要角色。虽然它的类名和Java层的Looper类一样，但此二者其实并无任何关系。这一点以后还将详细分析。

2.3.2 提取消息

当一切准备就绪后，Java层的消息循环处理，也就是Looper会在一个循环中提取并处理消息。消息的提取就是调用MessageQueue的next()方法。当消息队列为空时，next就会阻塞。MessageQueue同时支持Java层和Native层的事件，那么其next()方法该怎么实现呢？具体代码如下：

```
[MessagQueue.java-->MessageQueue.next()]
final Message next() {
    int pendingIdleHandlerCount = -1;
    int nextPollTimeoutMillis = 0;
    for (;;) {
        ......
        // mPtr保存了NativeMessageQueue的指针，调用nativePollOnce进行等待
        nativePollOnce(mPtr,nextPollTimeoutMillis);
        synchronized (this) {
            final long now = SystemClock.uptimeMillis();
            // mMessages用来存储消息，这里从其中取一个消息进行处理
            final Message msg = mMessages;
            if (msg != null) {
                final long when = msg.when;
                if (now &gt;= when) {
                    mBlocked = false;
                    mMessages = msg.next;
                    msg.next = null;
                    msg.markInUse();
                    return msg; // 返回一个Message给Looper进行派发和处理
               } else {
                    nextPollTimeoutMillis = (int) Math.min(when- now,
                                                  Integer.MAX_VALUE);
                }
            } else {
                nextPollTimeoutMillis = -1;
            }
            ......
            /* 处理注册的IdleHandler，当MessageQueue中没有Message时，
           Looper会调用IdleHandler做一些工作，例如做垃圾回收等  */
           ......
           pendingIdleHandlerCount = 0;
            nextPollTimeoutMillis = 0;
        }
    }
}
```

看到这里，可能会有人觉得这个MessageQueue很简单，不就是从以前在Java层的wait变成现在Native层的wait了吗？但是事情本质比表象要复杂得多，来思考下面的情况：

nativePollOnce()返回后，next()方法将从mMessages中提取一个消息。也就是说，要让nativePollOnce()返回，至少要添加一个消息到消息队列，否则nativePollOnce()不过是做了一次无用功罢了。

如果nativePollOnce()将在Native层等待，就表明Native层也可以投递Message，但是从Message类的实现代码上看，该类和Native层没有建立任何关系。那么nativePollOnce()在等待什么呢？

对于上面的问题，相信有些读者心中已有了答案：nativePollOnce()不仅在等待Java层来的Message，实际上还在Native还做了大量的工作。

下面我们来分析Java层投递Message并触发nativePollOnce工作的正常流程。

1. 在Java层投递Message

MessageQueue的enqueueMessage函数完成将一个Message投递到MessageQueue中的工作，其代码如下：

```
[MesssageQueue.java-->MessageQueue.enqueueMessage()]
final boolean enqueueMessage(Message msg, longwhen) {
    ......
    finalboolean needWake;
   synchronized (this) {
        if(mQuiting) {
           return false;
        }else if (msg.target == null) {
           mQuiting = true;
        }
       msg.when = when;
       Message p = mMessages;
        if(p == null || when == 0 || when &lt; p.when) {
           /* 如果p为空，表明消息队列中没有消息，那么msg将是第一个消息，needWake
             需要根据mBlocked的情况考虑是否触发 */
            msg.next= p;
           mMessages = msg;
           needWake = mBlocked;
        } else {
           // 如果p不为空，表明消息队列中还有剩余消息，需要将新的msg加到消息尾
           Message prev = null;
           while (p != null && p.when &lt;= when) {
               prev = p;
               p = p.next;
           }
           msg.next = prev.next;
           prev.next = msg;
           // 因为消息队列之前还剩余有消息，所以这里不用调用nativeWakeup
           needWake = false;
        }
    }
    if(needWake) {
        // 调用nativeWake，以触发nativePollOnce函数结束等待
       nativeWake(mPtr);
    }
    returntrue;
}
```

上面的代码比较简单，主要功能是：

- 将message按执行时间排序，并加入消息队。
- 根据情况调用nativeWake函数，以触发nativePollOnce函数，结束等待。

建议 虽然代码简单，但是对于那些不熟悉多线程的读者，还是要细细品味一下mBlocked值的作用。我们常说细节体现美，代码也一样，这个小小的mBlocked正是如此。

2. nativeWake函数分析

nativeWake函数的代码如下所示：

```
[android_os_MessageQueue.cpp-->android_os_MessageQueue_nativeWake()]
static voidandroid_os_MessageQueue_nativeWake(JNIEnv* env, jobject obj,
                                                      jint ptr)
{
   NativeMessageQueue* nativeMessageQueue = // 取出NativeMessageQueue对象
                       reinterpret_cast&lt;NativeMessageQueue*&gt;(ptr);
    returnnativeMessageQueue->wake(); // 调用它的wake函数
}
[android_os_MessageQueue.cpp-->NativeMessageQueue::wake()]
void NativeMessageQueue::wake() {
   mLooper->wake(); // 层层调用，现在转到mLooper的wake函数
}
```

Native Looper的wake函数代码如下：

```
[Looper.cpp-->Looper::wake()]
void Looper::wake() {
    ssize_tnWrite;
    do {
        // 向管道的写端写入一个字符
       nWrite = write(mWakeWritePipeFd, "W", 1);
    } while(nWrite == -1 && errno == EINTR);
}
```

Wake()函数则更为简单，仅仅向管道的写端写入一个字符"W"，这样管道的读端就会因为有数据可读而从等待状态中醒来。

2.3.3 nativePollOnce函数分析

nativePollOnce()的实现函数是android_os_MessageQueue_nativePollOnce，代码如下：

```
[android_os_MessageQueue.cpp-->android_os_MessageQueue_nativePollOnce()]
static voidandroid_os_MessageQueue_nativePollOnce(JNIEnv* env, jobject obj,
        jintptr, jint timeoutMillis)
    NativeMessageQueue* nativeMessageQueue =
                            reinterpret_cast&lt;NativeMessageQueue*&gt;(ptr);
    // 取出NativeMessageQueue对象，并调用它的pollOnce
   nativeMessageQueue->pollOnce(timeoutMillis);
}
```

分析pollOnce函数：

```
[android_os_MessageQueue.cpp-->NativeMessageQueue::pollOnece()]
void NativeMessageQueue::pollOnce(inttimeoutMillis) {
   mLooper->pollOnce(timeoutMillis); // 重任传递到Looper的pollOnce函数
}
```

Looper的pollOnce函数如下：

```
[Looper.cpp-->Looper::pollOnce()]
inline int pollOnce(int timeoutMillis) {
    returnpollOnce(timeoutMillis, NULL, NULL, NULL);
}
```

上面的函数将调用另外一个有4个参数的pollOnce函数，这个函数的原型如下：

```
int pollOnce(int timeoutMillis, int* outFd, int*outEvents, void** outData)
```

其中：

- timeOutMillis参数为超时等待时间。如果为-1，则表示无限等待，直到有事件发生为止。如果值为0，则无需等待立即返回。
- outFd用来存储发生事件的那个文件描述符 。
- outEvents用来存储在该文件描述符1上发生了哪些事件，目前支持可读、可写、错误和中断4个事件。这4个事件其实是从epoll事件转化而来。后面我们会介绍大名鼎鼎的epoll。
- outData用于存储上下文数据，这个上下文数据是由用户在添加监听句柄时传递的，它的作用和pthread_create函数最后一个参数param一样，用来传递用户自定义的数据。

另外，pollOnce函数的返回值也具有特殊的意义，具体如下：

- 当返回值为ALOOPER_POLL_WAKE时，表示这次返回是由wake函数触发的，也就是管道写端的那次写事件触发的。
- 返回值为ALOOPER_POLL_TIMEOUT表示等待超时。
- 返回值为ALOOPER_POLL_ERROR，表示等待过程中发生错误。
- 返回值为ALOOPER_POLL_CALLBACK，表示某个被监听的句柄因某种原因被触发。这时，outFd参数用于存储发生事件的文件句柄，outEvents用于存储所发生的事件。

上面这些知识是和epoll息息相关的。

提示 查看Looper的代码会发现，Looper采用了编译选项(即#if和#else)来控制是否使用epoll作为I/O复用的控制中枢。鉴于现在大多数系统都支持epoll，这里仅讨论使用epoll的情况。

1. epoll基础知识介绍

epoll机制提供了Linux平台上最高效的I/O复用机制，因此有必要介绍一下它的基础知识。

从调用方法上看，epoll的用法和select/poll非常类似，其主要作用就是I/O复用，即在一个地方等待多个文件句柄的I/O事件。

下面通过一个简单例子来分析epoll的工作流程。

```
/* **① 使用epoll前，需要先通过epoll_create函数创建一个epoll句柄。**
  下面一行代码中的10表示该epoll句柄初次创建时候分配能容纳10个fd相关信息的缓存。
  对于2.6.8版本以后的内核，该值没有实际作用，这里可以忽略。其实这个值的主要目的是
  确定分配一块多大的缓存。现在的内核都支持动态拓展这块缓存，所以该值就没有意义了 */
int epollHandle = epoll_create(10);
/* **② 得到epoll句柄后，下一步就是通过epoll_ctl把需要监听的文件句柄加入到epoll句柄中。**
  除了指定文件句柄本身的fd值外，同时还需要指定在该fd上等待什么事件。epoll支持四类事件，
  分别是EPOLLIN(句柄可读)、EPOLLOUT(句柄可写),EPOLLERR(句柄错误)、EPOLLHUP(句柄断)。
  epoll定义了一个结构体struct epoll_event来表达监听句柄的诉求。
  假设现在有一个监听端的socket句柄listener，要把它加入到epoll句柄中 */
struct epoll_event listenEvent; //先定义一个event
/* EPOLLIN表示可读事件,EPOLLOUT表示可写事件, 另外还有EPOLLERR,EPOLLHUP表示
  系统默认会将EPOLLERR加入到事件集合中 */
listenEvent.events = EPOLLIN;// 指定该句柄的可读事件
// epoll_event中有一个联合体叫data，用来存储上下文数据，本例的上下文数据就是句柄自己
listenEvent.data.fd = listenEvent;
  /* **③** EPOLL_CTL_ADD将监听fd和监听事件加入到epoll句柄的等待队列中；
    EPOLL_CTL_DEL将监听fd从epoll句柄中移除；
    EPOLL_CTL_MOD修改监听fd的监听事件，例如本来只等待可读事件，现在需要同时等待
    可写事件，那么修改listenEvent.events 为EPOLLIN|EPOLLOUT后，再传给epoll句柄*/
epoll_ctl(epollHandle,EPOLL_CTL_ADD,listener,&listenEvent);
/* 当把所有感兴趣的fd都加入到epoll句柄后，就可以开始坐等感兴趣的事情发生了。
  为了接收所发生的事情，先定义一个epoll_event数组 */
struct epoll_event resultEvents[10];
int timeout = -1;
while(1) {
    /* **④ 调用epoll_wait用于等待事件。**其中timeout可以指定一个超时时间，
     resultEvents用于接收发生的事件，10为该数组的大小。
     epoll_wait函数的返回值有如下含义：
      nfds大于0表示所监听的句柄上有事件发生；
      nfds等于0表示等待超时；
      nfds小于0表示等待过程中发生了错误*/
    int nfds = epoll_wait(epollHandle,resultEvents, 10, timeout);
    if(nfds == -1) {
        // epoll_wait发生了错误
    } else if(nfds == 0) {
        //发生超时，期间没有发生任何事件
    } else{
        // ⑤resultEvents用于返回那些发生了事件的信息
        for(int i = 0; i &lt; nfds; i++) {
            struct epoll_event & event =resultEvents[i];
            if(event & EPOLLIN) {
                /* **⑥ 收到可读事件。**到底是哪个文件句柄发生该事件呢？可通过event.data这个联合
                 体取得 前传递给epoll的上下文数据，该上下文信息可用于判断到底是谁发生了事件 */
              ......
            }
            .......//其他处理
        }
    }
}
```

epoll整体使用流程如上面代码所示，基本和select/poll类似，不过作为Linux平台最高效的I/O复用机制，这里有些内容供读者参考，

epoll的效率为什么会比select高？其中一个原因是调用方法。每次调用select时，都需要把感兴趣的事件复制到内核中，而epoll只在epll_ctl进行加入的时候复制一次。另外，epoll内部用于保存事件的数据结构使用的是红黑树，查找速度很快。而select采用数组保存信息，不但一次能等待的句柄个数有限，并且查找起来速度很慢。当然，在只等待少量文件句柄时，select和epoll效率相差不是很多，但还是推荐使用epoll。

epoll等待的事件有两种触发条件，一个是水平触发（EPOLLLEVEL），另外一个是边缘触发（EPOLLET,ET为Edge Trigger之意），这两种触发条件的区别非常重要。读者可通过man epoll查阅系统提供的更为详细的epoll机制。

最后，关于pipe，还想提出一个小问题供读者思考讨论：

为什么Android中使用pipe作为线程间通讯的方式？对于pipe的写端写入的数据，读端都不感兴趣，只是为了简单的唤醒。POSIX不是也有线程间同步函数吗？为什么要用pipe呢？

关于这个问题的答案，可参见邓凡平的一篇博文“随笔之如何实现一个线程池”。

- http://www.cnblogs.com/innost/archive/2011/11/24/2261454.html

2. pollOnce()函数分析

下面分析带4个参数的pollOnce()函数，代码如下：

```
[Looper.cpp-->Looper::pollOnce()]
int Looper::pollOnce(int timeoutMillis, int*outFd, int* outEvents,
void** outData) {
    intresult = 0;
    for(;;) { // 一个无限循环
        // mResponses是一个Vector，这里首先需要处理response
       while (mResponseIndex &lt; mResponses.size()) {
           const Response& response = mResponses.itemAt(mResponseIndex++);
           ALooper_callbackFunc callback = response.request.callback;
            if (!callback) {// 首先处理那些没有callback的Response
               int ident = response.request.ident; // ident是这个Response的id
               int fd = response.request.fd;
               int events = response.events;
               void* data = response.request.data;
               ......
               if (outFd != NULL) *outFd = fd;
               if (outEvents != NULL) *outEvents = events;
               if (outData != NULL) *outData = data;
               /* 实际上，对于没有callback的Response，pollOnce只是返回它的
                 ident，并没有实际做什么处理。因为没有callback，所以系统也不知道如何处理 */
               return ident;
           }
        }
        if(result != 0) {
            if(outFd != NULL) *outFd = 0;
           if (outEvents != NULL) *outEvents = NULL;
            if (outData != NULL) *outData = NULL;
           return result;
        }
        // 调用pollInner函数。注意，它在for循环内部
       result = pollInner(timeoutMillis);
    }
}
```

初看上面的代码，可能会让人有些丈二和尚摸不着头脑。但是把pollInner()函数分析完毕，大家就会明白很多。pollInner()函数非常长，把用于调试和统计的代码去掉，结果如下：

```
[Looper.cpp-->Looper::pollInner()]
int Looper::pollInner(int timeoutMillis) {
    if(timeoutMillis != 0 && mNextMessageUptime != LLONG_MAX) {
       nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
       ......//根据Native Message的信息计算此次需要等待的时间
       timeoutMillis = messageTimeoutMillis;
```

```
    }
    intresult = ALOOPER_POLL_WAKE;
   mResponses.clear();
   mResponseIndex = 0;
#ifdef LOOPER_USES_EPOLL  // 只讨论使用epoll进行I/O复用的方式
    structepoll_event eventItems[EPOLL_MAX_EVENTS];
    // 调用epoll_wait，等待感兴趣的事件或超时发生
    inteventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS,
                                  timeoutMillis);
#else
    ......//使用别的方式进行I/O复用
#endif
    //从epoll_wait返回，这时候一定发生了什么事情
   mLock.lock();
    if(eventCount &lt; 0) { //返回值小于零，表示发生错误
        if(errno == EINTR) {
           goto Done;
        }
        //设置result为ALLOPER_POLL_ERROR,并跳转到Done
       result = ALOOPER_POLL_ERROR;
        gotoDone;
    }
   //eventCount为零，表示发生超时，因此直接跳转到Done
    if(eventCount == 0) {
      result = ALOOPER_POLL_TIMEOUT;
        gotoDone;
    }
#ifdef LOOPER_USES_EPOLL
    // 根据epoll的用法，此时的eventCount表示发生事件的个数
    for (inti = 0; i &lt; eventCount; i++) {
        intfd = eventItems[i].data.fd;
       uint32_t epollEvents = eventItems[i].events;
        /* 之前通过pipe函数创建过两个fd，这里根据fd知道是管道读端有可读事件。
         读者还记得对nativeWake函数的分析吗？在那里我们向管道写端写了一个"W"字符，这样
         就能触发管道读端从epoll_wait函数返回了 */
        if(fd == mWakeReadPipeFd) {
           if (epollEvents & EPOLLIN) {
                // awoken函数直接读取并清空管道数据，读者可自行研究该函数
               awoken();
           }
            ......
        }else {
          /* mRequests和前面的mResponse相对应，它也是一个KeyedVector，其中存储了
           fd和对应的Request结构体，该结构体封装了和监控文件句柄相关的一些上下文信息，
            例如回调函数等。我们在后面的小节会再次介绍该结构体 */
           ssize_t requestIndex = mRequests.indexOfKey(fd);
           if (requestIndex &gt;= 0) {
               int events = 0;
               // 将epoll返回的事件转换成上层LOOPER使用的事件
               if (epollEvents & EPOLLIN) events |= ALOOPER_EVENT_INPUT;
               if (epollEvents & EPOLLOUT) events |= ALOOPER_EVENT_OUTPUT;
               if (epollEvents & EPOLLERR) events |= ALOOPER_EVENT_ERROR;
               if (epollEvents & EPOLLHUP) events |= ALOOPER_EVENT_HANGUP;
               // 每处理一个Request，就相应构造一个Response
               pushResponse(events, mRequests.valueAt(requestIndex));
           }
            ......
        }
    }
Done: ;
#else
     ......
#endif
    // 除了处理Request外，还处理Native的Message
   mNextMessageUptime = LLONG_MAX;
    while(mMessageEnvelopes.size() != 0) {
       nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
       const MessageEnvelope& messageEnvelope =mMessageEnvelopes.itemAt(0);
        if(messageEnvelope.uptime &lt;= now) {
           {
               sp&lt;MessageHandler&gt; handler = messageEnvelope.handler;
               Message message = messageEnvelope.message;
```

```
                mMessageEnvelopes.removeAt(0);
                mSendingMessage = true;
                mLock.unlock();
                /* 调用Native的handler处理Native的Message
                 从这里也可看出Native Message和Java层的Message没有什么关系 */
                handler->handleMessage(message);
            }
            mLock.lock();
            mSendingMessage = false;
            result = ALOOPER_POLL_CALLBACK;
         }else {
             mNextMessageUptime = messageEnvelope.uptime;
            break;
         }
     }
    mLock.unlock();
     // 处理那些带回调函数的Response
     for (size_t i = 0; i &lt; mResponses.size();i++) {
        const Response& response = mResponses.itemAt(i);
        ALooper_callbackFunc callback = response.request.callback;
         if(callback) {// 有了回调函数，就能知道如何处理所发生的事情了
            int fd = response.request.fd;
            int events = response.events;
            void* data = response.request.data;
            // 调用回调函数处理所发生的事件
            int callbackResult = callback(fd, events, data);
            if (callbackResult == 0) {
                // callback函数的返回值很重要，如果为0，表明不需要再次监视该文件句柄
                 removeFd(fd);
            }
            result = ALOOPER_POLL_CALLBACK;
         }
     }
     returnresult;
}
```

看完代码了，是否还有点模糊？那么，回顾一下pollInner函数的几个关键点：

- 首先需要计算一下真正需要等待的时间。
- 调用epoll_wait函数等待。
- epoll_wait函数返回，这时候可能有三种情况：

  a) 发生错误，则跳转到Done处。

  b) 超时，这时候也跳转到Done处。

  c) epoll_wait监测到某些文件句柄上有事件发生。

- 假设epoll_wait因为文件句柄有事件而返回，此时需要根据文件句柄来分别处理：

  a) 如果是管道读这一端有事情，则认为是控制命令，可以直接读取管道中的数据。

  b) 如果是其他FD发生事件，则根据Request构造Response，并push到Response数组中。

- 真正开始处理事件是在有Done标志的位置。

  a) 首先处理Native的Message。调用Native Handler的handleMessage处理该Message。

b) 处理Response数组中那些带有callback的事件。

上面的处理流程还是比较清晰的，但还是有个一个拦路虎，那就是mRequests，下面就来清剿这个拦路虎。

3. 添加监控请求

添加监控请求其实就是调用epoll_ctl增加文件句柄。下面通过从Native的Activity找到的一个例子来分析mRequests。

```
[android_app_NativeActivity.cpp-->loadNativeCode_native()]
static jint
loadNativeCode_native(JNIEnv* env, jobject clazz,jstring path,
                          jstringfuncName,jobject messageQueue,
                          jstringinternalDataDir, jstring obbDir,
                          jstringexternalDataDir, int sdkVersion,
                          jobject jAssetMgr,jbyteArray savedState)
{
    ......
    /* 调用Looper的addFd函数。第一个参数表示监听的fd；第二个参数0表示ident；
      第三个参数表示需要监听的事件，这里为只监听可读事件；第四个参数为回调函数，当该fd发生
      指定事件时，looper将回调该函数；第五个参数code为回调函数的参数 */
    code->looper->addFd(code->mainWorkRead,0,
                          ALOOPER_EVENT_INPUT,mainWorkCallback, code);
  ......
}
```

Looper的addFd()代码如下所示：

```
[Looper.cpp-->Looper::addFd()]
int Looper::addFd(int fd, int ident, int events,
                      ALooper_callbackFunccallback, void* data) {
    if (!callback) {
         /* 判断该Looper是否支持不带回调函数的文件句柄添加。一般不支持，因为没有回调函数
           Looper也不知道如何处理该文件句柄上发生的事情 */
         if(! mAllowNonCallbacks) {
           return -1;
        }
      ......
    }
#ifdef LOOPER_USES_EPOLL
    intepollEvents = 0;
    // 将用户的事件转换成epoll使用的值
    if(events & ALOOPER_EVENT_INPUT) epollEvents |= EPOLLIN;
    if(events & ALOOPER_EVENT_OUTPUT) epollEvents |= EPOLLOUT;
    {
       AutoMutex _l(mLock);
       Request request; // 创建一个Request对象
       request.fd = fd; // 保存fd
       request.ident = ident; // 保存id
       request.callback = callback; //保存callback
       request.data = data;  // 保存用户自定义数据
       struct epoll_event eventItem;
       memset(& eventItem, 0, sizeof(epoll_event));
       eventItem.events = epollEvents;
       eventItem.data.fd = fd;
        // 判断该Request是否已经存在，mRequests以fd作为key值
       ssize_t requestIndex = mRequests.indexOfKey(fd);
        if(requestIndex &lt; 0) {
           // 如果是新的文件句柄，则需要为epoll增加该fd
           int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, fd, &eventItem);
           ......
           // 保存Request到mRequests键值数组
           mRequests.add(fd, request);
        }else {
           // 如果之前加过，那么就修改该监听句柄的一些信息
           int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_MOD, fd, &eventItem);
          ......
           mRequests.replaceValueAt(requestIndex, request);
        }
    }
#else
    ......
#endif
    return1;
}
```

4. 处理监控请求

我们发现在pollInner()函数中，当某个监控fd上发生事件后，就会把对应的Request取出来调用。

```
pushResponse(events, mRequests.itemAt(i));
```

此函数如下：

```
[Looper.cpp-->Looper::pushResponse()]
void Looper::pushResponse(int events, constRequest& request) {
    Responseresponse;
   response.events = events;
   response.request = request; //其实很简单，就是保存所发生的事情和对应的Request
    mResponses.push(response);//然后保存到mResponse数组
}
```

根据前面的知识可知，并不是单独处理Request，而是需要先收集Request，等到Native Message消息处理完之后再做处理。这表明，在处理逻辑上，Native Message的优先级高于监控FD的优先级。

下面来了解如何添加Native的Message。

5. Native的sendMessage

Android 2.2中只有Java层才可以通过sendMessage()往MessageQueue中添加消息，从4.0开始，Native层也支持sendMessage()了。sendMessage()的代码如下：

```
[Looper.cpp-->Looper::sendMessage()]
void Looper::sendMessage(constsp&lt;MessageHandler&gt;& handler,
                              constMessage& message) {
    //Native的sendMessage函数必须同时传递一个Handler
    nsecs_tnow = systemTime(SYSTEM_TIME_MONOTONIC);
   sendMessageAtTime(now, handler, message); //调用sendMessageAtTime
}
[Looper.java-->Looper::sendMessageAtTime()]
void Looper::sendMessageAtTime(nsecs_t uptime,
                                     constsp&lt;MessageHandler&gt;& handler,
                                     constMessage& message) {
   size_t i= 0;
    {
       AutoMutex _l(mLock);
       size_t messageCount = mMessageEnvelopes.size();
        // 按时间排序，将消息插入到正确的位置上
       while (i &lt; messageCount &&
               uptime &gt;= mMessageEnvelopes.itemAt(i).uptime) {
           i += 1;
        }
       MessageEnvelope messageEnvelope(uptime, handler, message);
       mMessageEnvelopes.insertAt(messageEnvelope, i, 1);
        // mSendingMessage和Java层中的那个mBlocked一样，是一个小小的优化措施
        if(mSendingMessage) {
           return;
        }
    }
    // 唤醒epoll_wait，让它处理消息
    if (i ==0) {
       wake();
    }
}
```

2.3.4 MessageQueue总结

想不到，一个小小的MessageQueue竟然有如此多的内容。在后面分析Android输入系统时，会再次在Native层和MessageQueue碰面，这里仅是为后面的相会打下一定的基础。

现在将站在一个比具体代码更高的层次来认识一下MessageQueue和它的伙伴们。

1. 消息处理的大家族合照

MessageQueue只是消息处理大家族的一员，该家族的成员合照如图2-5所示。

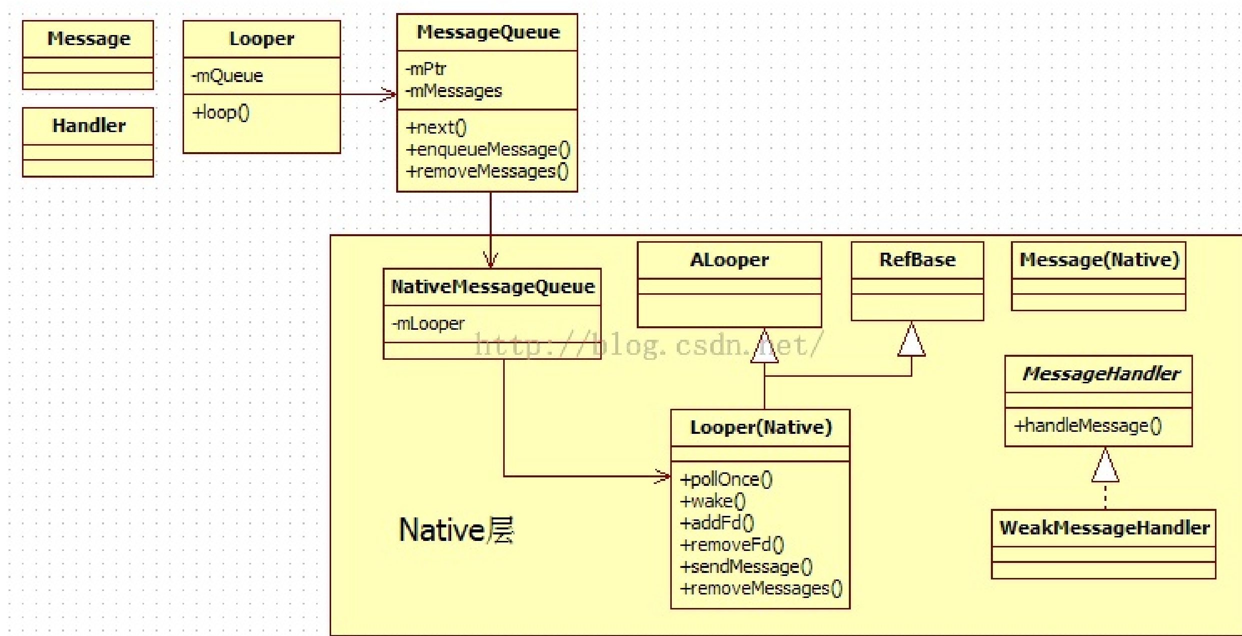


图 2 - 5 消息处理的家族合照

结合前述内容可从图2-5中得到：

- Java层提供了Looper类和MessageQueue类，其中Looper类提供循环处理消息的机制，MessageQueue类提供一个消息队列，以及插入、删除和提取消息的函数接口。另外，Handler也是在Java层常用的与消息处理相关的类。
- MessageQueue内部通过mPtr变量保存一个Native层的NativeMessageQueue对象，mMessages保存来自Java层的Message消息。
- NativeMessageQueue保存一个native的Looper对象，该Looper从ALooper派生，提供pollOnce和addFd等函数。
- Java层有Message类和Handler类，而Native层对应也有Message类和MessageHandler抽象类。在编码时，一般使用的是MessageHandler的派生类WeakMessageHandler类。

注意 在include/media/stagfright/foundation目录下也定义了一个ALooper类，它是供stagefright使用的类似Java消息循环的一套基础类。这种同名类的产生，估计是两个事先未做交流的Group的人写的。

2. MessageQueue处理流程总结

- MessageQueue核心逻辑下移到Native层后，极大地拓展了消息处理的范围，总结一下有以下几点：

- MessageQueue继续支持来自Java层的Message消息，也就是早期的Message加Handler的处理方式。
- MessageQueue在Native层的代表NativeMessageQueue支持来自Native层的Message，是通过Native的Message和MessageHandler来处理的。
- NativeMessageQueue还处理通过addFd添加的Request。在后面分析输入系统时，还会大量碰到这种方式。
- 从处理逻辑上看，先是Native的Message，然后是Native的Request，最后才是Java的Message。

2.4 本章小结

本章先对Java层的Binder架构做了一次较为深入的分析。Java层的Binder架构和Native层Binder架构类似，但是Java的Binder在通信上还是依赖Native层的Binder。建议想进一步了解Native Binder工作原理的读者，阅读卷I第6章“深入理解Binder”。另外，本章还对MessageQueue进行了较为深入的分析。Android 2.2中那个功能简单的MessageQueue现在变得复杂了，原因是该类的核心逻辑下移到Native层，导致现在的MessageQueue除了支持Java层的Message派发外，还新增了支持Native Message派发以及处理来自所监控的文件句柄的事件。

第3章 深入理解AudioService

本章主要内容：

- 探讨AudioService如何进行音量管理。
- 了解音频外设的管理机制。
- 探讨AudioFocus的工作原理。
- 介绍Android 4.1下AudioService的新特性。

本章涉及的源代码文件名及位置：

- AudioManager.java

framework\base\media\java\android\media\AudioManager.java

- AudioService.java

framework\base\media\java\android\media\AudioService.java

- AudioSystem.java

framework\base\media\java\android\media\AudioSystem.java

- VolumePanel.java

Framework\base\core\java\android\view\VolumePanel.java

- WiredAccessoryObserver.java

Framework\base\services\java\com\android\server\WiredAccessoryObserver.java

- PhoneWindow.java

Framework\base\policy\src\com\android\internal\policy\impl\PhoneWindow.java

- Activity.java

Framework\base\core\java\android\app\Activity.java

3.1概述

通过学习对《深入理解Android：卷I》（以后简称“卷I”）第7章的学习，相信大家已经对AudioTrack、AudioRecord、音频设备路由等知识有了深入的了解。这一章将详细介绍音频系统在Java层的实现，围绕AudioService这个系统服务深入探讨在Android SDK 中看到的音频

相关的机制的实现。

在分析Android音频系统时，习惯将其实现分为两个部分：数据流和策略。数据流描述了音频数据从数据源流向目的地的过程。而策略则是管理及控制数据流的路径与呈现的过程。在卷I所探讨的Native 层音频系统里，AudioTrack、AudioRecord和AudioFlinger可以被划归到数据流的范畴去讨论。而AudioPolicy相关的内容则属于策略范畴。

音频系统在Java层中基本上是不参与数据流的。虽然有AudioTrack和AudioRecord这两个类，但是他们只是Native层同名类的Java封装。抛开这两个类，AudioService这个系统服务包含或使用了几乎所的音频相关的内容，所以说AudioService是一个音频系统的大本营，它的功能非常多，而且它们之间的耦合性也不大，本章将从三个方面来探讨AudioService。

- 音量控制。
- 从按下音量键到弹出音量调提示框的过程，以及静音功能的工作原理。
- 音频IO设备的管理。

我们将详细探讨从插入耳机到声音经由耳机发出这个过程中，AudioService的工作内容。

- AudioFocus机制。

AudioService在2.3及以后版本中提供了AudioFocus机制用以结束多个音频应用混乱的交互现状。音频应用在播放音频的过程中需要合理的申请与释放AudioFocus，并根据AudioFocus所有权的变化来调整自己的播放行为。我们将从音频应用开始播放音频，到播放完成的过程中探讨AudioFocus的作用及原理。

AudioService的类图如下：

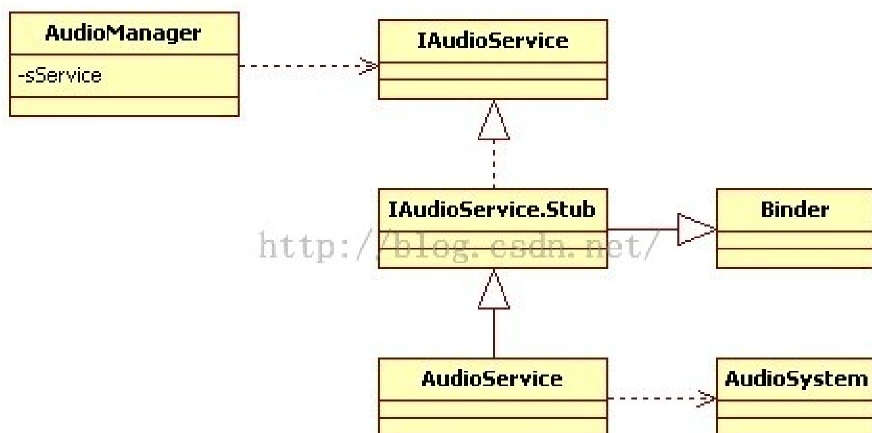


图 3?1 AudioService

由图3-1可知：

- AudioService继承自IAudioService.Stub。IAudioService.Stub类很明显是通过IAudioService.aidl自动生成的。AudioService位于Bn端。

- AudioManager拥有AudioService的Bp端，是AudioService在客户端的一个代理。几乎所有客户端对AudioManager进行的请求，最终都会交由AudioService实现。
- AudioService的功能实现依赖AudioSystem类，AudioSystem无法实例化，它是java层到native层的代理。AudioService将通过它与AudioPolicyService以及AudioFlinger进行交互。

那么，开始AudioService之旅吧。

3.2 音量管理

在Android手机上有两种改变系统音量的方式。最直接的做法就是通过手机的音量键进行音量调整，还有就是从设置界面中调整某一种类型音频的音量。另外，应用程序可以随时将某种类型的音频静音。他们都是都是通过AudioService进行的。

本节将从上述的三个方面对AudioService的音量管理进行探讨。

3.2.1音量键的处理流程

1. 触发音量键

音量键被按下后，Android输入系统将该事件一路派发给Activity，如果无人截获并消费这个事件，承载当前Activity的显示的PhoneWindow类的onKeyDown()或onKeyUp()函数将会将其处理，从而开始了通过音量键调整音量的处理流程。输入事件的派发机制以及PhoneWindow类的作用将在后续章节中详细介绍，现在只需要知道，PhoneWindow描述了一片显示区域，用于显示与管理我们所看到的Activity、对话框等内容。同时，它还是输入事件的派发对象，而且只有显示在最上面的PhoneWindow才会收到事件。

注意 按照Android的输入事件派发策略，Window对象在事件的派发队列中排在Activity的后面（应该说排在队尾比较合适），所以应用程序可以重写自己的onKeyDown()函数，将音量键用作其他的功能。比如说，在一个相机应用中，按下音量键所执行的动作是拍照而不是调节音量。

PhoneWindow的onKeyDown()函数实现如下：

```
[PhoneWindow.java-->PhoneWindow.onKeyDown()]
....../加省略号，    略过一些内容
switch (keyCode) {
    caseKeyEvent.KEYCODE_VOLUME_UP:
    caseKeyEvent.KEYCODE_VOLUME_DOWN:
case KeyEvent.KEYCODE_VOLUME_MUTE: {
    // 直接调用到AudioManager的handleKeyUp里面去了。是不是很简单而且直接呢
        getAudioManager().handleKeyDown(event,mVolumeControlStreamType);
       return true;
}
……
}
```

注意handleKeyDown()函数的第二个参数，它的意义是指定音量键将要改变哪一种流类型的音量。在Android中，音量的控制与流类型是密不可分的，每种流类型都独立地拥有自己的音量设置，绝大部分情况下互不干扰，例如音乐音量、通话音量就是相互独立的。所以说，离开流类型谈音量是没有意义的。在Android中，音量这个概念一定是描述的某一种流类型的音量。

这里传入了mVolumeControlStreamType，那么这个变量的值是从哪里来的呢？做过多媒体应用程序的读者应该知道，Activity类中有一个函数名为setVolumeControlStream(int streamType)。应用可以通过调用这个函数来指定显示这个Activity时音量键所控制的流类型。这个函数的内容很简单，就一行如下：

```
[Activity.java-->Activity.setVolumeControlStream()]
getWindow().setVolumeControlStream(streamType);
```

getWindow()的返回值的就是用于显示当前Activity的PhoneWindow。从名字就可以看出，这个调用改变了mVolumeControlStreamType，于是也就改变了按下音量键后传入AudioManager.handleKeyUp()函数的参数，从而达到了setVolumeControlStream的目的。同时，还应该能看出，这个设置是被绑定到Activity的Window上的，不同Activity之间切换时，接受按键事件的Window也会随之切换，所以应用不需要去考虑在其生命周期中音量键所控制的流类型的切换问题。

AudioManager的handleKeyDown()的实现很简单，在一个switch中，它调用了AudioService的adjustSuggestedStreamVolume()，所以直接看一下AudioService的这个函数。

2. adjustSuggestedStreamVolume()分析

我们先来看函数原型，

```
public voidadjustSuggestedStreamVolume(int direction,
                  int suggestedStreamType,
                   int flags)
```

adjustSuggestedStreamVolume()有三个参数，而第三个参数flags的意思就不那么容易猜了。其实AudioManager在handleKeyDown()里设置了两个flags，分别是FLAG_SHOW_UI和FLAG_VIBRATE。从名字上我们就能看出一些端倪。前者用于告诉AudioService我们需要弹出一个音量控制面板。而在handleKeyUp()里设置了FLAG_PLAY_SOUND，这是为什么当松开音量键后“有时候”会有一个提示音。注意，handleKeyUp()设置了FLAG_PLAY_SOUND，但是只是有时候这个flag才会生效，我们在下面的代码中能看到为什么。还须要注意的是，第二个参数名为suggestedStreamType，从其命名来推断，这个参数传入的流类型对于AudioService来说只是一个建议，是否采纳这个建议AudioService则有自己的考虑。

```
[AudioService.java-->AudioService.adjustSuggestedStreamVolume()]
public void adjustSuggestedStreamVolume(intdirection, int suggestedStreamType,
                           int flags) {格式要调整好
int streamType;
// ①从这一小段代码中，可以看出在 AudioService中还有地方可以强行改变音量键控制的流类型
    if(mVolumeControlStream != -1) {
       streamType = mVolumeControlStream;
} else {
     // ②通过getActiveStreamType()函数获取要控制的流类型
     // 这里根据建议的流类型与AudioService的实际情况，返回一个值
       streamType = getActiveStreamType(suggestedStreamType);
    }
// ③这个啰嗦的if判断的目的，就是只有在特定的流类型下,并且没有处于锁屏状态时才会播放声音
    if((streamType != STREAM_REMOTE_MUSIC) &&
           (flags & AudioManager.FLAG_PLAY_SOUND) != 0 &&
           ((mStreamVolumeAlias[streamType] != AudioSystem.STREAM_RING)
            || (mKeyguardManager != null &&mKeyguardManager.isKeyguardLocked()))) {
        flags&= ~AudioManager.FLAG_PLAY_SOUND;
    }
    if(streamType == STREAM_REMOTE_MUSIC) {
…… //我们不讨论远程播放的情况
} else {
    // ④调用adjustStreamVolume
       adjustStreamVolume(streamType, direction, flags);
    }
}
```

注意 初看着段代码时，可能有读者会对下面这句话感到疑惑：

```
VolumeStreamState streamState =mStreamStates[mStreamVolumeAlias[streamType]];
```

其实这是为了满足所谓的"将铃声音量用作通知音量"这种需求。这样就需要实现在两个有这个需求的流A与B之间建立起一个A→B映射。当我们对A流进行音量操作时，实际上是在操作B流。其实笔者个人认为这个功能对用户体验的提升并不大，但是却给AudioService的实现增加了不小的复杂度。直观上来想，我们可能想使用一个HashMap解决这个问题，键是源流类型，值目标流类型。而Android使用了一个更简单那但是却不是那么好理解的方法来完成这件事。AudioService用一个名为mStreamVolumeAlias的整形数组来描述这个映射关系。

如果想要实现"以铃声音量用作音乐音量"，只需要修改相应位置的值为STREAM_RING即可，就像下面这样：

```
mStreamVolumeAlias[AudioSystem.STREAM_MUSIC] =AudioSystem.STREAM_RING;
```

之后，因为需求要求对A流进行音量操作时，实际上是在操作B流，所以就不难理解为什么在很多和流相关的函数里都会先做这样的一个转换：

```
streamType = mStreamVolumeAlias[streamType];
```

其具体的工作方式就留给读者进行思考了。在本章的分析过程中，大可忽略这种转换，这并不影响我们对音量控制原理的理解。

这个函数简单来说，做三件事：

- 确定要调整音量的流类型。
- 在某些情况下屏蔽FLAG_PLAY_SOUND。
- 调用adjustStreamVolume()。

关于这个函数仍然有几点需要说明一下。它刚开始的时候有一个判断，条件是一个名为mVolumeControlStream的整型变量是否等于-1，从这块代码来看，mVolumeControlStream比参数传入的suggestedStreamType厉害多了，只要它不是-1，那么要调整音量的流类型就是它。那这么厉害的控制手段，是做什么用的呢？其实，mVolumeControlStream是VolumePanel通过forceVolumeControlStream()函数设置的。什么是VolumePanel呢？就是我们按下音量键后的那个音量条提示框了。VolumePanel在显示时会调用forceVolumeControlStream强制后续的音量键操作固定为促使它显示的那个流类型。并在它关闭时取消这个强制设置，即置mVolumeControlStream为-1。这个我们在后面分析VolumePanel时会看到。

接下来我们继续看一下adjustStreamVolume()的实现。

3. adjustStreamVolume()分析

```
[AudioService.java-->AudioService.adjustStreamVolume()]
public void adjustStreamVolume(int streamType, intdirection, int flags) {
// 首先还是获取streamType映射到的流类型。这个映射的机制确实给我们的分析带来不小的干扰
// 在非必要的情况下忽略它们吧
int streamTypeAlias = mStreamVolumeAlias[streamType];
// 注意VolumeStreamState类
   VolumeStreamState streamState = mStreamStates[streamTypeAlias];
    final intdevice = getDeviceForStream(streamTypeAlias);
    // 获取当前音量，注意第二个参数的值，它的目的是如果这个流被静音，则取出它被静音前的音量
    final intaliasIndex = streamState.getIndex(device,
                                         (streamState.muteCount()!= 0)
    booleanadjustVolume = true;
// rescaleIndex用于将音量值的变化量从源流类型变换到目标流类型下
// 由于不同的流类型的音量调节范围不同，所以这个转换是必需的
    int step= rescaleIndex(10, streamType, streamTypeAlias);
//上面准备好了所需的所有信息，接下来要做一些真正有用的动作了
// 比如说checkForRingerModeChange()。调用这个函数可能变更情景模式
// 它的返回值adjustVolume是一个布尔变量，用来表示是否有必要继续设置音量值
// 这是因为在一些情况下，音量键用来改变情景模式，而不是设置音量值
    if(((flags & AudioManager.FLAG_ALLOW_RINGER_MODES) != 0) ||
           (streamTypeAlias == getMasterStreamType())) {
……
       adjustVolume = checkForRingerModeChange(aliasIndex, direction, step);
……
    }
int index;
// 取出调整前的音量值。这个值稍后被用在sendVolumeUpdate()的调用中
    final intoldIndex = mStreamStates[streamType].getIndex(device,
           (mStreamStates[streamType].muteCount() != 0) /* lastAudible */);
// 接下来我们可以看到，只有流没有被静音时，才会设置音量到底层去，否则只调整其静音前的音量
// 为了简单起见，暂不考虑静音时的情况
    if(streamState.muteCount() != 0) {
        ……
} else {
    // 为什么还要判断streamState.adjustIndex的返回值呢?
    // 因为如果音量值在adjust之后并没有发生变化，比如说达到了最大值，就不需要继续后面的操作了
        if(adjustVolume && streamState.adjustIndex(direction * step, device)) {
           // 发送消息给AudioHandler
           // 这个消息在setStreamVolumeInt()函数的分析中已经看到过了
           // 这个消息将把音量设置到底层去，并将其存储到SettingsProvider中去
           sendMsg(mAudioHandler,
                   MSG_SET_DEVICE_VOLUME,
                   SENDMSG_QUEUE,
                   device,
                   0,
                   streamState,
                   0);
        }
        index= mStreamStates[streamType].getIndex(device, false  /* lastAudible */);
    }
    // 最后，调用sendVolumeUpdate函数，通知外界音量值发生了变化
   sendVolumeUpdate(streamType, oldIndex, index, flags);
}
```

在这个函数的实现中，有一个非常重要的类型：VolumeStreamState。前面我们提到过，Android的音量是依赖于某种流类型的。如果Android定义了N个流类型，AudioService就需要维护N个音量值与之对应。另外每个流类型的音量等级范围不一样，所以还需要为每个流类型维护他们的音量调节范围。VolumeStreamState类的功能就是为了保存了一个流类型所有音量相关的信息。AudioService为每一种流类型都分配了一个VolumeStreamState对象，并以流类型的值为索引，保存在一个名为数组mStreamStates中。在这个函数中调用了VolumeStreamState对象的adjustIndex()函数，于是就改变了这个对象中存储的音量值。不过，仅仅是改变了它的存储值，并没有把这个变化设置到底层。

总结一下这个函数都作了什么：

- 准备工作。计算按下音量键的音量步进值。细心的读者一定注意到了，这个步进值是10而不是1。原来，在VolumeStreamState中保存的音量值是其实际值的10倍。为什么这么做呢？这是为了在不同流类型之间进行音量转换时能够保证一定精度的一种奇怪的实现，其转换过程读者可以参考rescaleIndex()函数的实现。我们可以将这种做法理解为在转换过程中保留了小数点后一位的精度。其实，直接使用float类型来保存岂不是更简单呢？
- 检查是否需要改变情景模式。checkForRingerModeChange()和情景模式有关。读者可以自行研究其实现。
- 调用adjustIndex()更改VolumeStreamState对象中保存的音量值。
- 通过sendMsg()发送消息MSG_SET_DEVICE_VOLUME到mAudioHandler。
- 调用sendVolumeUpdate()函数，通知外界音量发生了变化。

我们将重点分析后面三个内容：adjustIndex()、MSG_SET_DEVICE_VOLUME消息的处理和sendVolumeUpdate()。

4. VolumeStreamState的adjustIndex()分析

我们看一下这个函数的定义：

```
[AudioService.java-->VolumeStreamState.adjustIndex()]
public boolean adjustIndex(int deltaIndex, intdevice) {
// 将现有的音量值加上变化量，然后调用setIndex设置下去
// 返回值与setIndex一样
    return setIndex(getIndex(device,  false /* lastAudible */) + deltaIndex,
                              device,
                              true  /* lastAudible */);
}
```

这个函数很简单，我们再看一下setIndex()的实现：

```
[AudioService.java-->VolumeStreamState.setIndex()]
public synchronized boolean setIndex(int index, intdevice, boolean lastAudible) {
    intoldIndex = getIndex(device, false  /*lastAudible */);
    index =getValidIndex(index);
    // 在VolumeStreamState中保存设置的音量值，注意是用了一个HashMap
   mIndex.put(device, getValidIndex(index));
    if(oldIndex != index) {
        // 保存到lastAudible
        if(lastAudible) {
           mLastAudibleIndex.put(device, index);
        }
        // 同时设置所有映射到当前流类型的其他流的音量
       boolean currentDevice = (device == getDeviceForStream(mStreamType));
        intnumStreamTypes = AudioSystem.getNumStreamTypes();
        for(int streamType = numStreamTypes - 1; streamType &gt;;= 0; streamType--) {
            ……
        }
       return true;
    } else {
       return false;
    }
}
```

在这个函数中有三个工作要做：

- 首先是保存设置的音量值，这是VolumeStreamState的本职工作，这和4.1之前的版本不一样，音量值与设备相关联了。于是对于同一种流类型来说，在不同的音频设备下将会拥有不同的音量值。
- 然后就是根据参数的要求保存音量值到mLastAudibleIndex里面去。从名字就可以看出，它保存了静音前的音量。当取消静音时，AudioService就会恢复到这里保存的音量。
- 再就是对流映射的处理。既然A->B，那么设置B的音量时，同时要改变A的音量。这就是后面那个循环的作用。

可以看出，VolumeStreamState.adjustIndex()除了更新自己所保存的音量值外，没有做其他的事情，接下来就看一下MSG_SET_DEVICE_VOLUME的消息处理做了什么。

5. MSG_SET_DEVICE_VOLUME消息的处理

adjustStreamVolume()函数使用sendMsg()函数发送了MSG_SET_DEVICE_VOLUME消息给了mAudioHandler，这个Handler运行在AudioService的主线程上。直接看一下在mAudioHandler中负责处理MSG_SET_DEVICE_VOLUME消息的setDeviceVolume()函数：

```
[AudioService.java-->AudioHandler.setIndex()]
private void setDeviceVolume(VolumeStreamStatestreamState, int device) {
// 调用VolumeStreamState的applyDeviceVolume。
// 这个函数的内容很简单，就是在调用AudioSystem.setStreamVolumeIndex()
// 到这里，音量就被设置到底层的AudioFlinger里面去了
   streamState.applyDeviceVolume(device);
    // 和上面一样，需要处理流音量映射的情况。这段代码和上面setIndex的相关代码很像，不是么
    intnumStreamTypes = AudioSystem.getNumStreamTypes();
for (int streamType = numStreamTypes - 1; streamType &gt;= 0;streamType--) {
     ……
        }
    }
     // 发送消息给mAudioHandler，其处理函数将会调用persitVolume()函数这将会把音量的   //设置信息存
// AudioService在初始化时，将会从SettingsProvider中将音量设置读取出来并进行设置
   sendMsg(mAudioHandler,
           MSG_PERSIST_VOLUME,
           SENDMSG_QUEUE,
           PERSIST_CURRENT|PERSIST_LAST_AUDIBLE,
           device,
           streamState,
           PERSIST_DELAY);
}
```

注意 sendMsg()是一个异步的操作，这就意味着，完成adjustIndex()更新音量信息后adjustStreamVolume()函数就返回了，但是音量并没有立刻地被设置到底层。而且由于Handler处理多个消息的过程是串行的，这就隐含着一个风险：当Handler正在处理某一个消息时发生了阻塞，那么当按下音量键时，调用adjustStreamVolume()虽然可以立刻返回，而且从界面上看或者用getStreamVolume()获取音量值发现都是没有问题的，但是手机发出声音时的音量大小并没有改变。

6. sendVolumeUpdate()分析

接下来，分析一下sendVolumeUpdate()函数，它用于通知外界音量发生了变化。

```
 [AudioService.java-->AudioService.sendVolumeUpdate()]
private void sendVolumeUpdate(int streamType, intoldIndex, int index, int flags) {
    // 读者可能会对这句话感到有点奇怪，mVoiceCapable是从SettingsProvider中取出来的一个常量
    // 从某种意义上来说，它可以用来判断设备是否拥有通话功能。对于没有通话能力的设备来说，RING流类
    // 型自然也就没有意义了。这句话应该算是一种从语义操作上进行的保护
    if(!mVoiceCapable && (streamType == AudioSystem.STREAM_RING)) {
       streamType = AudioSystem.STREAM_NOTIFICATION;
    }
    //mVolumePanel是一个VolumePanel类的实例，就是它显示了音量提示框
   mVolumePanel.postVolumeChanged(streamType, flags);
    // 发送广播。可以看到它们都有(x+5)/10的一个操作。为什么要除以10可以理解，但是+5的意义呢
    // 原来是为了实现四舍五入
    oldIndex= (oldIndex + 5) / 10;
    index =(index + 5) / 10;
    Intentintent = new Intent(AudioManager.VOLUME_CHANGED_ACTION);
   intent.putExtra(AudioManager.EXTRA_VOLUME_STREAM_TYPE, streamType);
   intent.putExtra(AudioManager.EXTRA_VOLUME_STREAM_VALUE, index);
   intent.putExtra(AudioManager.EXTRA_PREV_VOLUME_STREAM_VALUE, oldIndex);
   mContext.sendBroadcast(intent);
}
```

这个函数将音量的变化通过广播的形式通知给了其他感兴趣得模块。同时，它还特别通知了mVolumePanel。mVolumePanel是VolumePanel类的一个实例。我们所看到的音量调节通知框就是它了。

至此，从按下音量键开始的整个处理流程就完结了。在继续分析音量调节通知框的工作原李之前，先对之前的分析过程作一个总结，请参考下面的序列图：

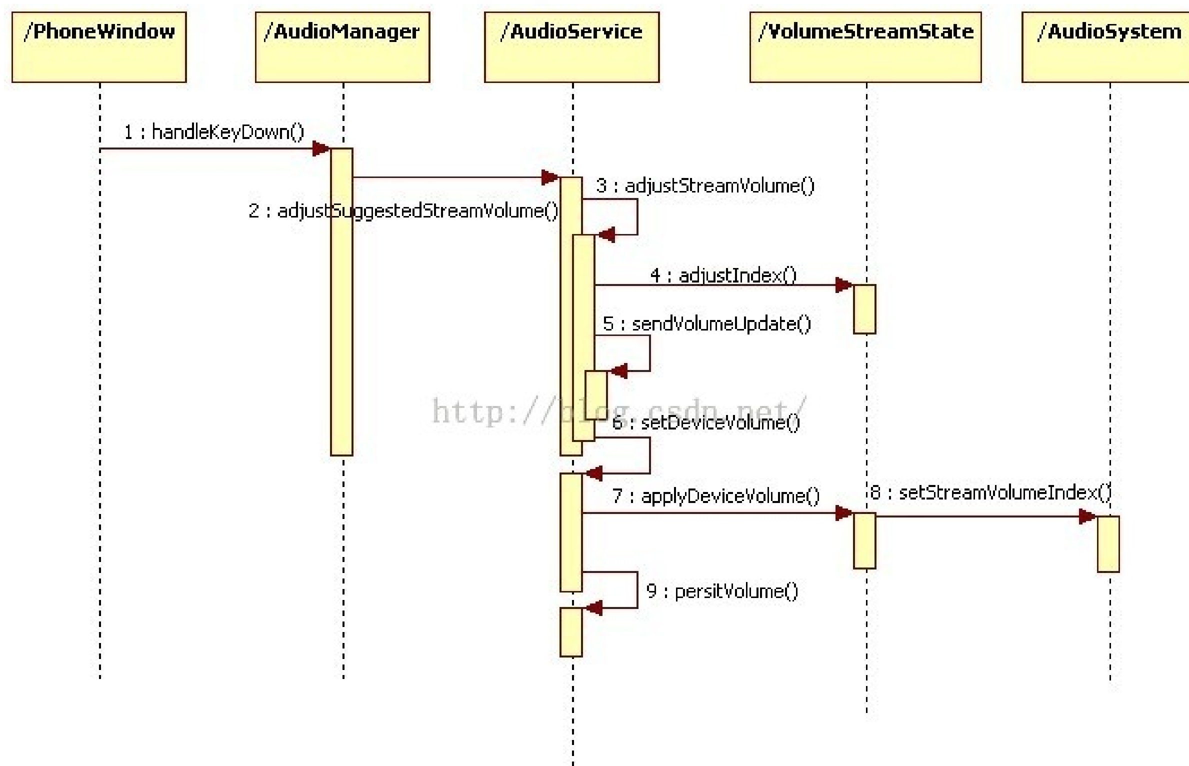


图 3-2 音量键调整音量的处理流程

结合上面分析的结果，由图 3-2可知：

- 音量键处理流程的发起者是PhoneWindow。
- AudioManager仅仅起到代理的作用。
- AudioService接受AudioManager的调用请求，操作VolumeStreamState的实例进行音量的设置。
- VolumeStreamState负责保存音量设置，并且提供了将音量设置到底层的方法。
- AudioService负责将设置结果以广播的形式通知外界。

到这里，相信大家对音量量调节的流程已经有了一个比较清晰的认识了。接下来我们将介绍音量调节通知框的工作原理。

4. 音量调节通知框的工作原理

在分析sendVolumeUpdate()函数时曾经注意到它调用了mVolumePanel的postVolumeChanged()函数。mVolumePanel是一个VolumePanel的实例。作为一个Handler的子类，它承接了音量变化的UI/声音的通知工作。在继续上面的讨论之前，先了解一下其工作的基本原理。

VolumePanel为于android.view包下，但是却没有在API中被提供。因为它只能被AudioService使用，所以和AudioService放在一个包下可能更合理一些。从这个类的注释上可以看到，谷歌的开发人员对它被放在android.view下也有极大的不满（What A Mass! 他们这么写道……）。

VolumePanel下定义了两个重要的子类型，分别是StreamResources和StreamControl。StreamResources实际上是一个枚举。它的每一个可用元素保存了一个流类型的通知框所需要的各种资源，如图标、提示文字等等。其定义就像下面这样：

```
[VolumePanel.java-->VolumePanel.StreamResources]
private enum StreamResources {
   BluetoothSCOStream(AudioManager.STREAM_BLUETOOTH_SCO,
           R.string.volume_icon_description_bluetooth,
           R.drawable.ic_audio_bt,
           R.drawable.ic_audio_bt,
           false),
    // 后面的几个枚举项我们省略了其构造参数，与BluetoothSCOStream的内容是一致的
   RingerStream(……),
   VoiceStream(……),
   AlarmStream(……),
   MediaStream(……),
   NotificationStream(……),
   MasterStream(……),
    RemoteStream(……);
    intstreamType; // 流类型
    intdescRes;     // 描述信息
    inticonRes;     // 图标
    inticonMuteRes;// 静音图标
    booleanshow;    // 是否显示
    StreamResources(intstreamType, int descRes, int iconRes, int iconMuteRes, boolean sho
        ……
    }
};
```

这几个枚举项组成了一个数组名为STREAM如下：

```
[VolumePanel.java-->VolumePanel.STREAMS]
private static final StreamResources[] STREAMS = {
   StreamResources.BluetoothSCOStream,
   StreamResources.RingerStream,
   StreamResources.VoiceStream,
   StreamResources.MediaStream,
   StreamResources.NotificationStream,
   StreamResources.AlarmStream,
   StreamResources.MasterStream,
   StreamResources.RemoteStream
};
```

VolumePanel将从这个STREAMS数组中获取它所支持的流类型的相关资源。这么做是不是觉得有点啰嗦呢？事实上，在这里使用枚举并没有什么特殊的意义，使用普通的一个Java类来定义StreamResources就已经足够了。

```
StreamControl类则保存了一个流类型的通知框所需要显示的控件。其定义如下:
[VolumePanel.java-->VolumePanel.StreamControl]
private class StreamControl {
    intstreamType;
    ViewGroupgroup;
    ImageViewicon;
    SeekBarseekbarView;
    inticonRes;
    inticonMuteRes;
}
```

很简单对不对？StreamControl实例中保存了音量条提示框中所需的所用控件。关于这个类在VolumePanel的使用，我们可能很直观的认为只有一个StreamControl实例，在对话框显示时，使其保存的控件按需加载指定流类型的StreamResources实例中定义的资源。其实不然，应该是出于对运行效率的考虑，StreamControl实例也是每个流类型人手一份，和StreamResources实例形成了一个一一对应的关系。所有的StreamControl 实例被保存在了一个以流类型的值为键的Hashtable中，名为mStreamControls。我们可以在StreamControl的初始化函数createSliders()中一窥其端倪：

```
[VolumePanel-->VolumePanel.createSliders()]
private void createSliders() {
    ……
    // 遍历STREAM中所有的StreamResources实例
    for (inti = 0; i &lt; STREAMS.length; i++) {
       StreamResources streamRes = STREAMS[i];
        intstreamType = streamRes.streamType;
        ……
        // 为streamType创建一个StreamControl
       StreamControl sc = new StreamControl();
        // 这里将初始化sc的成员变量
        ……
        // 将初始化好的sc放入mStreamControls中去。
       mStreamControls.put(streamType, sc);
    }
}
```

值得一提的是，这个初始化的工作并没有在构造函数中进行，而是在postVolumeChanged()函数里处理的。

既然已经有了通知框所需要的资源和通知框的控件了，那么接下来就要有一个对话框承载它们。没错，VolumePanel保存了一个名为mDialog的Dialog实例，这就是通知框的本尊了。每当有新的音量变化到来时，mDialog的内容就会被替换为制定流类型对应的StreamControl中所保存的控件，并根据音量变化情况设置其音量条的位置，最后调用mDialog.show()显示出来。同时，发送一个延时消息MSG_TIMEOUT，这条延时消息生效时，将会关闭提示框。

StreamResource、StreamControl与mDialog的关系就像下面这附图一样，StreamControl可以说是mDialog的配件，随需拆卸。

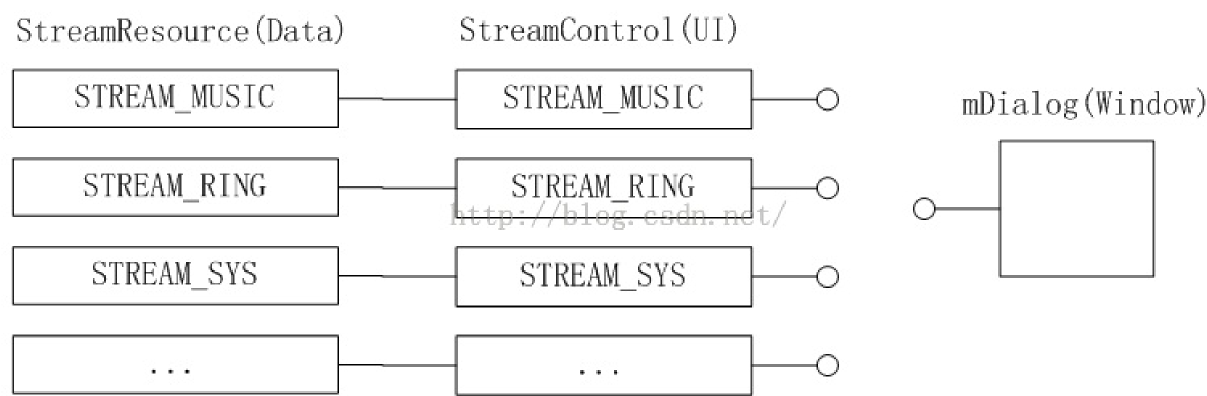


图 3-3 StreamResource、StreamControl与mDialog的关系

接下来具体看一下VolumePanel在收到音量变化通知后都做了什么。我们在上一小节中说到了mVolumePanel.postVolumeChanged()函数。它的内容很简单，直接发送了一条消息MSG_VOLUME_CHANGED，然后在handleMessage中调用onVolumeChanged()函数进行真正的处理。

注意 VolumePanel在MSG_VOLUME_CHANGED的消息处理函数中调用onVolumeChanged()函数而不直接在postVolumeChanged()函数中直接调，。这么做是有实际意义的。由于Android要求只能在创建控件的线程中对控件进行操作。postVolumeChanged()作为一个回调性质的函数，不能要求调用者位于哪个线程中。所以必须通过向Handler发送消息的方式，将后续的操作转移到指定的线程中去。在大家设计具有UI Controller功能的类时，VolumePanel的实现方式有很好的参考意义。

看一下onVolumeChanged()函数的实现：

```
[VolumePanel.java-->VolumePanel.onVolumeChanged()]
protected void onVolumeChanged(int streamType, intflags) {
    //  需要flags中包含AudioManager.FLAG_SHOW_UI才会显示音量调通知框
    if((flags & AudioManager.FLAG_SHOW_UI) != 0) {
       synchronized (this) {
           if (mActiveStreamType != streamType) {
               reorderSliders(streamType); // 在Dialog里装载需要的StreamControl
            }
           // 这个函数负责最终的显示
           onShowVolumeChanged(streamType, flags);
        }
    }
    // 是否要播出Tone音，注意有个小延迟
    if((flags & AudioManager.FLAG_PLAY_SOUND) != 0 && ! mRingIsSilent) {
       removeMessages(MSG_PLAY_SOUND);
       sendMessageDelayed(obtainMessage(MSG_PLAY_SOUND, streamType, flags),PLAY_SOUND_DEL
    }
    // 取消声音与振动的播放
    if((flags & AudioManager.FLAG_REMOVE_SOUND_AND_VIBRATE) != 0) {
       removeMessages(MSG_PLAY_SOUND);
       removeMessages(MSG_VIBRATE);
       onStopSounds();
    }
    // 开始安排回收资源
   removeMessages(MSG_FREE_RESOURCES);
   sendMessageDelayed(obtainMessage(MSG_FREE_RESOURCES), FREE_DELAY);
    // 重置音量框超时关闭的时间。
   resetTimeout();
}
```

注意最后一个resetTimeout()的调用。它其实是重新延时发送了MSG_TIMEOUT消息。当MSG_TIMEOUT消息生效时，mDialog将会被关闭。

之后就是onShowVolumeChanged了。这个函数负责为通知框的内容填充音量、图表等信息，然后再把通知框显示出来，如果还没有显示的话。以铃声音量为例，省略掉其他的代码。

```
[VolumePanel.java-->VolumePanel.onShowVolumeChanged()]
    protectedvoid onShowVolumeChanged(int streamType, int flags) {
        // 获取音量值
        intindex = getStreamVolume(streamType);
        // 获取音量最大值，这两个将用来设置进度条
        intmax = getStreamMaxVolume(streamType);
       switch (streamType) {
           // 这个switch语句中，我们要根据每种流类型的特点，进行各种调整。
           // 例如Music就有时就需要更新它的图标，因为使用蓝牙耳机时的图标和和平时的不一样
           // 所以每一次都需要更新一下
           case AudioManager.STREAM_MUSIC: {
               // Special case for when Bluetooth is active for music
               if ((mAudioManager.getDevicesForStream(AudioManager.STREAM_MUSIC) &
                       (AudioManager.DEVICE_OUT_BLUETOOTH_A2DP |
                        AudioManager.DEVICE_OUT_BLUETOOTH_A2DP_HEADPHONES|
                       AudioManager.DEVICE_OUT_BLUETOOTH_A2DP_SPEAKER)) != 0) {
                   setMusicIcon(R.drawable.ic_audio_bt,
                                    R.drawable.ic_audio_bt_mute);// 设置蓝牙图标
               } else {
                   setMusicIcon(R.drawable.ic_audio_vol,
                                    R.drawable.ic_audio_vol_mute);//设置为普通图标
               }
               break;
            }
            ……
        }
        // 取出Music流类型对应的StreamControl。并设置其SeekBar的音量显示
       StreamControl sc = mStreamControls.get(streamType);
        if(sc != null) {
           if (sc.seekbarView.getMax() != max) {
               sc.seekbarView.setMax(max);
            }
           sc.seekbarView.setProgress(index);
            ……
        }
        if(!mDialog.isShowing()) {  //  如果对话框还没有显示
            /* forceVolumeControlStream()的调用在这里，一旦此通知框被显示，之后的按下音量键，都只能
           mAudioManager.forceVolumeControlStream(streamType);
           // 为Dialog设置显示控件
           // 注意,mView目前已经在reorderSlider()函数中安装好了Music流所对应的
           //StreamControl了
           mDialog.setContentView(mView);
            ……
            //显示对话框
           mDialog.show();
        }
}
```

至此，音量条提示框就被显示出来了。总结一下它的工作过程：

- postVolumeChanged() 是VolumePanel显示的入口。
- 检查flags中是否有FLAG_SHOW_UI。
- VolumePanel会在第一次被要求弹出时初始化其控件资源。
- mDialog 加载指定流类型对应的StreamControl，也就是控件。
- 显示对话框，并开始超时计时。
- 超时计时到达，关闭对话框。

到此为止，AudioService对音量键的处理流程就介绍完了。而 Android还有另外一种改变音量的方式。

3.2.2通用的音量设置函数setStreamVolume()

除了通过音量键可以调节音量以外，用户还可以在系统设置中进行调节。AudioManager.setStreamVolume()是系统设置界面中调整音量所使用的接口。

1. setStreamVolume()分析

setStreamVolume()是SDK中提供给应用的API，它的作用是为特定的流类型设置范围内允许的任意音量。我们看一下它的实现：

```
[AudioService.java-->AudioService.setStreamVolume()]
public void setStreamVolume(int streamType, intindex, int flags) {
    // 这里先判断一下流类型这个参数的有效性
    ensureValidStreamType(streamType);
    // 获取保存了指定流类型音量信息的VolumeStreamState对象。
    // 注意这里面使用mStreamVolumeAlias对这个数组进行了流类型的转换
   VolumeStreamState streamState =mStreamStates[mStreamVolumeAlias[streamType]];
    // 获取当前流将使用哪一个音频设备进行播放。它最终会调用到AudioPolicyService里去
    final intdevice = getDeviceForStream(streamType);
    // 获取流当前的音量
    final intoldIndex = streamState.getIndex(device,
                        (streamState.muteCount()!= 0) /* lastAudible */);
    // 将原流类型下的音量值映射到目标流类型下的音量值
    // 因为不同流类型的音量值刻度不一样，所以需要进行这个转换
    index =rescaleIndex(index * 10, streamType, mStreamVolumeAlias[streamType]);
    //暂时先忽略下面这段if中的代码。它的作用根据flags的要求修改手机的情景模式
    if(((flags & AudioManager.FLAG_ALLOW_RINGER_MODES) != 0) ||
           (mStreamVolumeAlias[streamType] == getMasterStreamType())) {
        ……
    }
    // 调用setStreamVolumeInt()
   setStreamVolumeInt(mStreamVolumeAlias[streamType], index, device, false,true);
    // 获取设置的结果
    index =mStreamStates[streamType].getIndex(device,
(mStreamStates[streamType].muteCount() != 0) /*lastAudible */);
    // 广播通知
   sendVolumeUpdate(streamType, oldIndex, index, flags);
}
```

看明白这个函数了吗？抛开被忽略掉的那个if块归纳一下：它的工作其实很简单的，就是执行下面这三方面的工作：

- 为调用setStreamVolumeInt准备参数。
- 调用setStreamVolumeInt。
- 广播音量发生变化的通知。

分析的主线将转向setStreamVolumeInt()的内容了。

2. setStreamVolumeInt()分析

看一下setStreamVolumeInt函数的代码，和往常一样，暂时忽略目前与分析目标无关的部分代码。

```
[AudioService.java-->AudioService.setStreamVolumeInt()]
private void setStreamVolumeInt(int streamType,
                                int index,
                                int device,
                                boolean force,
                                booleanlastAudible) {
    // 获取保存音量信息的VolumeStreamState对象
   VolumeStreamState streamState = mStreamStates[streamType];
    if(streamState.muteCount() != 0) {
        // 这里的内容是为了处理当流已经被静音后的情况。我们在讨论静音的实现时在考虑这段代码
        ……
    } else {
        // 调用streamState.setIndex()
        if(streamState.setIndex(index, device, lastAudible) || force) {
           // 如果setIndex返回true或者force参数为true的话就在这里给mAudioHandler
           //
           sendMsg(mAudioHandler,
                   MSG_SET_DEVICE_VOLUME,
                   SENDMSG_QUEUE,
                   device,
                   0,
                   streamState,
                   0);
        }
    }
}
```

此函数有两个工作，一个是streamState.setIndex() 另一个则是根据setIndex()的返回值和force参数决定是否要发送MSG_SET_DEVICE_VOLUME消息。这两个内容在3.2.1节中已经又介绍了。在此不再赘述。

其执行过程可以参考下面的序列图：

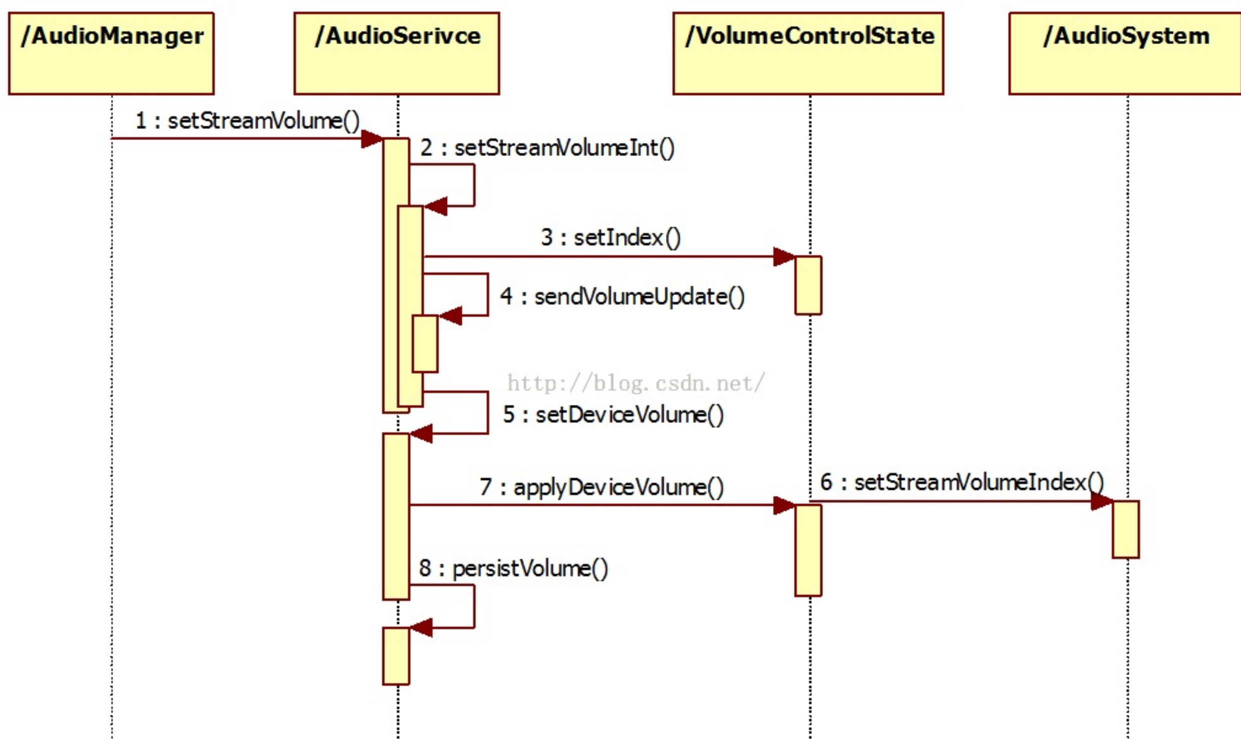


图 3?3setStreamVolume的处理流程

注意 看到这个序列图后，是否有读者感到眼熟呢？如果我们把setStreamVolumeInt()的内容替换掉在setStreamVolume()的对它的调用，再和adjustStreamVolume()函数进行以下比较，就会发现他们的内容出奇地相似。Android在其他地方也有这样的情况出现。从这一点上来说，已经发展到4.1版本的Android源码仍然尚不够精致。读者可以思考一下，有没有办法把这两个函数融合为一个函数呢？

到此，对于音量设置相关的内容就告一段落。接下来我们将讨论和音量相关的另一个重要的内容——静音。

3.2.3静音控制

静音控制的情况与音量调节又很大的不同。因为每个应用都有可能进行静音操作，所以为了防止状态发生紊乱，就需要为静音操作进行计数，也就是说多次静音后需要多次取消静音才可以。

不过，如果进行了静音计数后还会引入另外一个问题。如果一个应用在静音操作（计数加1）后因为某种原因不小心挂了，那么将不会有人再为它进行取消静音的操作，静音计数无法再回到0，也就是说这个倒霉的流将被永远静音下去。

那么怎么处理应用异常退出后的静音计数呢？AudioService的解决办法是记录下来每个应用的自己的静音计数，当应用崩溃时，在总的静音计数中减去崩溃应用自己的静音计数，也就是说，由我们为这个应用完成它没能完成的取消静音这个操作。为此，VolumeStreamState定义了一个继承自DeathRecepient的内部类名为VolumeDeathHandler，并为每个进行静音操作的进程创建一个实例。它保存了对应进程的静音计数，并在进程死亡时进行计数清零的操作。从这个名字来看可能是Google希望这个类将来能够承担更多与音量相关的事情吧，不过眼下它只负责静音。我们将在后续的内容对这个类进行深入的讲解。

经过前面的介绍，我们不难得出AudioService、VolumeStreamState与VolumeDeathHandler的关系如下：

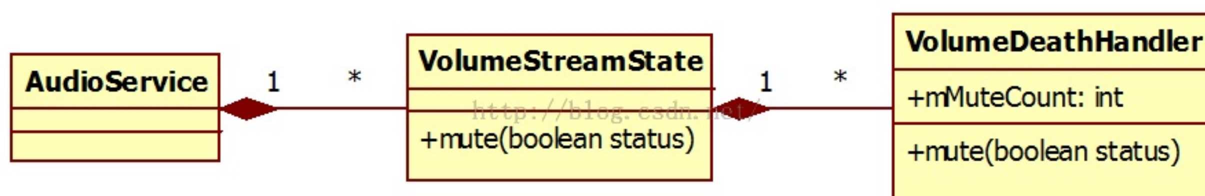


图 3?4 与静音相关的类

1. setStreamMute()分析

同音量设置一样，静音控制也是相对于某一个流类型而言的。而且正如本节开头所提到的，静音控制涉及到引用计数和客户端进程的死亡监控。所以相对与音量控制来说，静音控制有一定的复杂度。不过还好，静音控制对外入口只有一个函数，就是AudioManager.setStreamMute()。第二个参数state为true表示静音，否则为解除静音。

```
[AudioManager.java-->AudioManager.setStreamMute()]
public void setStreamMute(int streamType, booleanstate) {
   IAudioService service = getService();
try {
    // 调用AudioService的setStreamMute，注意第三个参数mICallBack。
       service.setStreamMute(streamType, state, mICallBack);
    } catch(RemoteException e) {
       Log.e(TAG, "Dead object in setStreamMute", e);
    }
}
```

AudioManager一如既往地充当着一个AudioService代理的一个角色。但是这次有一个小小的却很重要的动作。AudioManager给AudioService传入了一个名为mICallBack的变量。查看一下它的定义：

private final IBinder mICallBack = new Binder();

真是简单得不得了。全文搜索一下，我们发现它只被用来作为AudioService的几个函数调用的参数。从AudioManager这边看来它没有任何实际意义。其实，这在Android中进程间交互通讯中是一种常见且非常重要的技术。mICallBack这个简单的变量可以充当Bp端在Bn端的一个唯一标识。Bn端，也就是AudioService拿到这个标识后，就可以通过DeathRecipient机制获取到Bp端异常退出的回调。这是AudioService维持静音状态正常变迁的一个基石。

注意 服务端把客户端传入的这个Binder对象作为客户端的一个唯一标识，能做的事情不仅仅DeathRecipient这一个。还以这个标识为键创建一个Hashtable，用来保存每个客户端相关信息。这在Android各个系统服务的实现中是一种很常见的用法。

另外，本例中传入的mICallBack是直接从Binder类实例化出来的，是一个很原始的IBinder对象。进一步讲，如果传递了一个通过AIDL定义的IBinder对象，这个对象就有了交互能力，服务端可以它向客户端进行回调。在后面探讨AudioFocus机制时会遇到这种情况。

2. VolumeDeathHandler分析

我们继续跟踪AudioService.setStreamMute()的实现，记得注意第三个参数cb，它是代表特定客户端的标识。

```
[AudioService.java-->AudioService.setStreamMute()]
public void setStreamMute(int streamType, booleanstate, IBinder cb) {
    // 只有可以静音的流类型才能执行静音操作。这说明，并不是所有的流都可以被静音
    if(isStreamAffectedByMute(streamType)) {
        // 直接调用了流类型对应的mStreamStates的mute()函数
        // 这里没有做那个令人讨厌的流类型的映射。这是出于操作语义上的原因。读者可以自行思考一下
       mStreamStates[streamType].mute(cb, state);
    }
}
```

接下来是VolumeStreamState的mute()函数。VolumeStreamState的确是音量相关操作的核心类型。

```
[AudioService.java-->VolumeStreamState.mute()]
public synchronized void mute(IBinder cb, booleanstate) {
    // 这句话是一个重点，VolumeDeathHandler与cb一一对应
    // 用来管理客户端的静音操作，并且监控客户端的生命状态
   VolumeDeathHandler handler = getDeathHandler(cb, state);
    if(handler == null) {
        Log.e(TAG, "Could not get client deathhandler for stream: "+mStreamType);
       return;
    }
    // 通过VolumeDeathHandler执行静音操作
   handler.mute(state);
}
```

上述代码引入了静音控制的主角，VolumeDeathHandler，也许叫做MuteHandler更合适一些。它其实只有两个成员变量，分别是mICallBack和mMuteCount。其中mICallBack保存了客户端的传进来的标识，mMuteCount则保存了当前客户端执行静音操作的引用计数。另外，它继承自IBinder.DeathRecipient，所以它拥有监听客户端生命状态的能力。而成员函数则只有两个，分别是mute()和binderDied()。说到这里，再看看上面VolumeStreamState.mute()的实现，读者能否先想想VolumeDeathHandler的具体实现是什么样子的么？

继续上面的脚步，看一下它的mute()函数。它的参数state的取值指定了进行静音还是取消静音。所以这个函数也就分成了两部分，分别处理静音与取消静音两个操作。其实，这完全可以放在两个函数中完成。先看看静音操作是怎么做的吧。

```
[AudioService.java-->VolumeDeathHandler.mute()part1]
public void mute(boolean state) {
if (state) {
    // 静音操作
        if(mMuteCount == 0) {
           // 如果mMuteCount等于0，则表示客户端是第一次执行静音操作
           //此时我们linkToDeath，开始对客户端的生命状况进行监听
           //这样做的好处是可以避免非静音状态下对Binder资源的额外占用
           try {
               // linkToDeath! 为什么要判断是否为空？AudioManager不是写死了会把一个有效的
               // Binder传递进来么？原来AudioManager也可能会调用mute()
               // 此时的mICallback为空
               if (mICallback != null) {
                   mICallback.linkToDeath(this, 0);
               }
               // 保存的mDeathHandlers列表中去
               mDeathHandlers.add(this);
               // muteCount() 我们在后面会介绍，这是全局的静音操作的引用计数
               // 如果它的返回值为0，则表示这个流目前还没有被静音
               if (muteCount() == 0) {
                   // 在这里设置流的音量为0
                  ......//你打出来的省略号咋这么小呢？^_^
               }
            }catch (RemoteException e) {
                   ......
            }
        }
        // 引用计数加1
       mMuteCount++;
} else {
    // 暂时先不看取消静音的操作
……
    }
}
```

看明白了么？这个函数的条件嵌套比较多，仔细归纳一下，就会发现这段代码的思路是非常清晰的。静音操作根据条件满足与否，有三个任务要做：

- 无论什么条件下，只要执行了这个函数，静音操作的引用计数都会加1。
- 如果这是客户端第一次执行静音，则开始监控其生命状态，并把自己加入到VolumeStreamState的mDeathHandlers列表中去。这是这段代码中很精练的一个操作,只有在客户端执行过静音操作后才会对其生命状态感兴趣，才有保存其VolumeDeathHandler的必要。
- 更进一步的，如果这是这个流类型第一次被静音，则设置流音量为0，这才是真正的静音动作。

不得不说，这段代码是非常精练的，不是说代码量少，而是它的行为非常干净。决不会做多余的操作，也不会保存多余的变量。

下面我们要看一下取消静音的操作。取消静音作为静音的逆操作，相信读者已经可以想象得到取消静音都做什么事情了吧？我们就不再对其进行说明了。

```
[AudioService.java-->VolumeDeathHandler.mute()part 2]
public void mute(boolean state) {
if (state) {
    // 忽略掉静音操作
  ......
    } else {
        if(mMuteCount == 0) {
           Log.e(TAG, "unexpected unmute for stream: "+mStreamType);
        }else {
           // 引用计数减1先
           mMuteCount--;
           if (mMuteCount == 0) {
// 如果这是客户端最后一次有效地取消静音
               mDeathHandlers.remove(this);
               if (mICallback != null) {
                   mICallback.unlinkToDeath(this, 0);
               }
               if (muteCount() == 0) {
                   // 将流的音量值设置回静音前的音量，也就是lastAudibleIndex
……
               }
            }
        }
    }
}
```

然后就剩下最后的binderDied()函数了。当客户端发生异常，没能取消其执行过的静音操作时，需要替它完成它应该做却没做的事情。

```
[AudioService.java-->VolumeDeathHandler.binderDied()]
public void binderDied() {
    if(mMuteCount != 0) {
       mMuteCount = 1;
       mute(false);
    }
}
```

这个实现不难理解。读者可以将自行分析一下为什么这么做可以消除意外退出的客户端遗留下来的影响。

3.2.4 音量控制总结

音量控制是AudioService最重要的功能之一。经过上面的讨论，相信读者对AudioService的音量管理流程已经有了一定的理解。

总结一下我们在这一节里所学到的内容：

- AudioService音量管理的核心是VolumeStreamState。它保存了一个流类型所有的音量信息。
- VolumeStreamState保存了运行时的音量信息，而音量的生效则是在底层AudioFlinger完成。所以音量设置需要做两件事情，更新VolumeStreamState存储的音量值。设置音量到Audio底层系统。

- VolumeDeathHandler是VolumeStreamState的一个内部类。它的实例对应了在一个流类型上执行了静音操作的一个客户端，是实现静音功能的核心对象。

第4章 深入理解WindowManagerService（节选）

本章主要内容：

- 示例最原始最简单的窗口创建方法
- 研究WMS的窗口管理结构
- 探讨WMS布局系统的工作原理
- 研究WMS动画系统的工作原理

本章涉及的源代码文件名及位置：

- SystemServer.java

frameworks/base/services/java/com/android/server/SystemServer.java

- WindowManagerService.java

frameworks/base/services/java/com/android/server/wm/WindowManagerService.java

- ActivityStack.java

frameworks/base/services/java/com/android/server/am/ActivityStack.java

- WindowState.java

frameworks/base/services/java/com/android/server/wm/WindowState.java

- PhoneWindowManager.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java

- AccelerateDecelerateInterpolator.java

frameworks/base/core/java/android/view/animation/AccelerateDecelerateInterpolator.java

- Animation.java

frameworks/base/core/java/android/view/animation/Animation.java

- AlphaAnimation.java

frameworks/base/core/java/android/view/animation/AlphaAnimation.java

- WindowAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowAnimator.java

- WindowStateAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowStateAnimator.java

4.1 初识WindowManagerService

WindowManagerService（以下简称WMS）是继ActivityManagerService与PackageManagerService之后又一个复杂却十分重要的系统服务。

在介绍WMS之前，首先要了解窗口（Window）是什么。

Android系统中的窗口是屏幕上的一块用于绘制各种UI元素并可以响应应用户输入的一个矩形区域。从原理上来讲，窗口的概念是独自占有一个Surface实例的显示区域。例如Dialog、Activity的界面、壁纸、状态栏以及Toast等都是窗口。

《卷I》第8章曾详细介绍了一个Activity通过Surface来显示自己的过程：

- Surface是一块画布，应用可以随心所欲地通过Canvas或者OpenGL在其上作画。
- 然后通过SurfaceFlinger将多块Surface的内容按照特定的顺序（Z-order）进行混合并输出到FrameBuffer，从而将Android“漂亮的脸蛋”显示给用户。

既然每个窗口都有一块Surface供自己涂鸦，必然需要一个角色对所有窗口的Surface进行协调管理。于是，WMS便应运而生。WMS为所有窗口分配Surface，掌管Surface的显示顺序（Z-order）以及位置尺寸，控制窗口动画，并且还是输入系统的一重要的中转站。

说明一个窗口拥有显示和响应用户输入这两层含义，本章将侧重于分析窗口的显示，而响应用户输入的过程则在第5章进行详细的介绍。

本章将深入分析WMS的两个基础子系统的工作原理：

- 布局系统（Layout System），计算与管理窗口的位置、层次。
- 动画系统（Animation System），根据布局系统计算的窗口位置与层次渲染窗口动画。

为了让读者对WMS的功能以及工作方式有一个初步地认识，并见识一下WMS的强大，本节将从一个简单而神奇的例子开始WMS的学习之旅。

4.1.1 一个从命令行启动的动画窗口

1. SampleWindow的实现

在这一节里将编写一个最简单的Java程序SampleWindow，仅使用WMS的接口创建并渲染一个动画窗口。此程序将抛开Activity、Wallpaper等UI架构的复杂性，直接了当地揭示WMS的客户端如何申请、渲染并注销自己的窗口。同时这也初步地反应了WMS的工作方式。

这个例子很简单，只有三个文件:

- SampleWindow.java 主程序源代码。
- Android.mk 编译脚本。
- sw.sh 启动器。

分别看一下这三个文件的实现：

```
[-->SampleWindow.java::SampleWindow]
package understanding.wms.samplewindow;
......
public class SampleWindow {
    publicstatic void main(String[] args) {
        try {
             //SampleWindow.Run()是这个程序的主入口
            new SampleWindow().Run();
        } catch (Exception e) {
           e.printStackTrace();
        }
    }
    //IWindowSession 是客户端向WMS请求窗口操作的中间代理，并且是进程唯一的
   IWindowSession mSession = null;
    //InputChannel 是窗口接收用户输入事件的管道。在第5章中将对其进行详细的探讨
   InputChannel mInputChannel = new InputChannel();
    // 下面的三个Rect保存了窗口的布局结果。其中mFrame表示了窗口在屏幕上的位置与尺寸
    // 在4.4中将详细介绍它们的作用以及计算原理
    RectmInsets = new Rect();
    RectmFrame = new Rect();
    RectmVisibleInsets = new Rect();
   Configuration mConfig = new Configuration();
    // 窗口的Surface，在此Surface上进行的绘制都将在此窗口上显示出来
    SurfacemSurface = new Surface();
    // 用于在窗口上进行绘图的画刷
    PaintmPaint = new Paint();
    // 添加窗口所需的令牌，在4.2节将会对其进行介绍
    IBindermToken = new Binder();
    // 一个窗口对象，本例演示了如何将此窗口添加到WMS中，并在其上进行绘制操作
    MyWindowmWindow = new MyWindow();
    //WindowManager.LayoutParams定义了窗口的布局属性，包括位置、尺寸以及窗口类型等
   LayoutParams mLp = new LayoutParams();
   Choreographer mChoreographer = null;
    //InputHandler 用于从InputChannel接收按键事件做出响应
   InputHandler mInputHandler = null;
    booleanmContinueAnime = true;
    publicvoid Run() throws Exception{
        Looper.prepare();
        // 获取WMS服务
       IWindowManager wms = IWindowManager.Stub.asInterface(
                      ServiceManager.getService(Context.WINDOW_SERVICE));
        // 通过WindowManagerGlobal获取进程唯一的IWindowSession实例。它将用于向WMS
        // 发送请求。注意这个函数在较早的Android版本（如4.1）位于ViewRootImpl类中
        mSession= WindowManagerGlobal.getWindowSession(Looper.myLooper());
        // 获取屏幕分辨率
       IDisplayManager dm = IDisplayManager.Stub.asInterface(
                        ServiceManager.getService(Context.DISPLAY_SERVICE));
       DisplayInfo di = dm.getDisplayInfo(Display.DEFAULT_DISPLAY);
       Point scrnSize = new Point(di.appWidth, di.appHeight);
        // 初始化WindowManager.LayoutParams
        initLayoutParams(scrnSize);
        // 将新窗口添加到WMS
        installWindow(wms);
        // 初始化Choreographer的实例，此实例为线程唯一。这个类的用法与Handler
        // 类似，不过它总是在VSYC同步时回调，所以比Handler更适合做动画的循环器[1]
        mChoreographer= Choreographer.getInstance();
        // 开始处理第一帧的动画
        scheduleNextFrame();
        // 当前线程陷入消息循环，直到Looper.quit()
        Looper.loop();
```

```
            // 标记不要继续绘制动画帧
            mContinueAnime= false;
            // 卸载当前Window
            uninstallWindow(wms);
        }
        publicvoid initLayoutParams(Point screenSize) {
            // 标记即将安装的窗口类型为SYSTEM_ALERT，这将使得窗口的ZOrder顺序比较靠前
           mLp.type = LayoutParams.TYPE_SYSTEM_ALERT;
            mLp.setTitle("SampleWindow");
            // 设定窗口的左上角坐标以及高度和宽度
           mLp.gravity = Gravity.LEFT | Gravity.TOP;
           mLp.x = screenSize.x / 4;
           mLp.y = screenSize.y / 4;
           mLp.width = screenSize.x / 2;
           mLp.height = screenSize.y / 2;
            // 和输入事件相关的Flag，希望当输入事件发生在此窗口之外时，其他窗口也可以接受输入事件
           mLp.flags = mLp.flags | WindowManager.LayoutParams.FLAG_NOT_TOUCH_MODAL;
        }
        publicvoid installWindow(IWindowManager wms) throws Exception {
            // 首先向WMS声明一个Token，任何一个Window都需要隶属与一个特定类型的Token
            wms.addWindowToken(mToken,WindowManager.LayoutParams.TYPE_SYSTEM_ALERT);
            // 设置窗口所隶属的Token
           mLp.token = mToken;
            // 通过IWindowSession将窗口安装进WMS，注意，此时仅仅是安装到WMS，本例的Window
            // 目前仍然没有有效的Surface。不过，经过这个调用后，mInputChannel已经可以用来接受
            // 输入事件了
            mSession.add(mWindow,0, mLp, View.VISIBLE, mInsets, mInputChannel);
            /*通过IWindowSession要求WMS对本窗口进行重新布局，经过这个操作后，WMS将会为窗口
              创建一块用于绘制的Surface并保存在参数mSurface中。同时，这个Surface被WMS放置在
            LayoutParams所指定的位置上 */
            mSession.relayout(mWindow,0, mLp, mLp.width, mLp.height, View.VISIBLE,
                            0, mFrame, mInsets,mVisibleInsets, mConfig, mSurface);
            if(!mSurface.isValid()) {
                thrownew RuntimeException("Failed creating Surface.");
            }
            // 基于WMS返回的InputChannel创建一个Handler，用于监听输入事件
            //mInputHandler一旦被创建，就已经在监听输入事件了
            mInputHandler= new InputHandler(mInputChannel, Looper.myLooper());
        }
        publicvoid uninstallWindow(IWindowManager wms) throws Exception {
            // 从WMS处卸载窗口
            mSession.remove(mWindow);
            // 从WMS处移除之前添加的Token
            wms.removeWindowToken(mToken);
        }
        publicvoid scheduleNextFrame() {
            // 要求在显示系统刷新下一帧时回调mFrameRender，注意，只回调一次
            mChoreographer.postCallback(Choreographer.CALLBACK_ANIMATION
                   , mFrameRender, null);
        }
        // 这个Runnable对象用以在窗口上描绘一帧
        publicRunnable mFrameRender = new Runnable() {
           @Override
            publicvoid run() {
                try{
                   // 获取当期时间戳
                   long time = mChoreographer.getFrameTime() % 1000;
                   // 绘图
                   if (mSurface.isValid()) {
                       Canvas canvas = mSurface.lockCanvas(null);
                       canvas.drawColor(Color.DKGRAY);
                       canvas.drawRect(2 * mLp.width * time / 1000
                                - mLp.width, 0, 2 *mLp.width * time
                                / 1000, mLp.height,mPaint);
                       mSurface.unlockCanvasAndPost(canvas);
                       mSession.finishDrawing(mWindow);
                   }
                if(mContinueAnime)
                   scheduleNextFrame();
               } catch (Exception e) {
                   e.printStackTrace();
               }
```

```
        }
    };
    // 定义一个类继承InputEventReceiver，用以在其onInputEvent()函数中接收窗口的输入事件
    classInputHandler extends InputEventReceiver {
       Looper mLooper = null;
        publicInputHandler(InputChannel inputChannel, Looper looper) {
            super(inputChannel,looper);
            mLooper= looper;
        }
       @Override
        publicvoid onInputEvent(InputEvent event) {
            if(event instanceof MotionEvent) {
               MotionEvent me = (MotionEvent)event;
                if (me.getAction() ==MotionEvent.ACTION_UP) {
                   // 退出程序
                   mLooper.quit();
               }
           }
            super.onInputEvent(event);
        }
    }
    // 实现一个继承自IWindow.Stub的类MyWindow。
    classMyWindow extends IWindow.Stub {
        // 保持默认的实现即可
    }
}
```

由于此程序使用了大量的隐藏API（即SDK中没有定义这些API），因此需要放在Android源码环境中进行编译它。对应的Android.mk如下：

```
[-->Android.mk]
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES := $(call all-subdir-java-files)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := samplewindow
include $(BUILD_JAVA_LIBRARY)
```

将这两个文件放在$TOP/frameworks/base/cmds/samplewindow/下，然后用make或mm命令进行编译。最终生成的结果是samplewindow.jar，文件位置在out/target/<ProductName>/system/framework/下。将该文件通过adb push到手机的/system/framework/下。

提示读者可使用Android4.2模拟器来运行此程序。

然而，samplewindow.jar不是一个可执行程序，。故，需借助Android的app_process工具来加载并执行它。笔者编写了一个脚本做为启动器：

```
[-->sw.sh]
base=/system
export CLASSPATH=$base/framework/samplewindow.jar
exec app_process $base/binunderstanding.wms.samplewindow.SampleWindow "$@"
```

注意 app_process其实就是大名鼎鼎的zygote。不过，只有使用--zygote参数启动时它才会给改名为zygote[2]，否则就像java –jar命令一样，运行指定类的main静态函数。

在手机中执行该脚本，其运行结果是一个灰色的方块不断地从屏幕左侧移动到右侧，如图4-1所示。

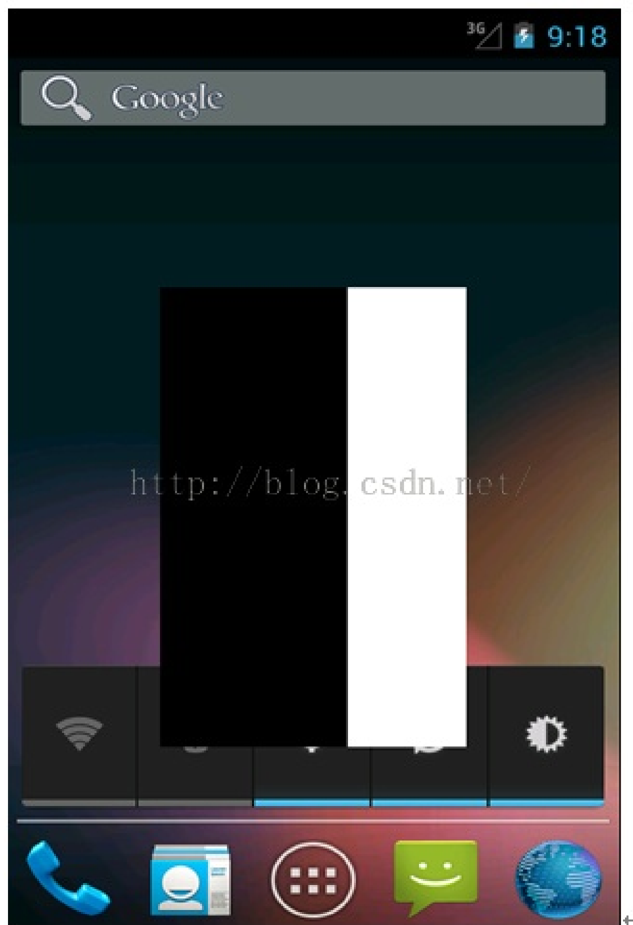


图 4-1 SampleWindow在手机中的运行效果

2. 初识窗口的创建、绘制与销毁

SampleWindow的这段代码虽然简单，但是却很好地提炼了一个窗口的创建、绘制以及销毁的过程。注意，本例没有使用任何 WMS以外的系统服务，也没有使用Android系统四大组件的框架，也就是说，如果你愿意，可以利用WMS实现自己的UI与应用程序框架，这样就可以衍生出一个新的平台了。

总结在客户端创建一个窗口的步骤：

- 获取IWindowSession和WMS实例。客户端可以通过IWindowSession向WMS发送请求。
- 创建并初始化WindowManager.LayoutParams。注意这里是WindowManager下的LayoutParams，它继承自ViewGroup.LayoutParams类，并扩展了一些窗口相关的属性。其中最重要的是type属性。这个属性描述了窗口的类型，而窗口类型正是WMS对多个窗口进行ZOrder排序的依据。

- 向WMS添加一个窗口令牌（WindowToken）。本章后续将分析窗口令牌的概念，目前读者只要知道，窗口令牌描述了一个显示行为，并且WMS要求每一个窗口必须隶属于某一个显示令牌。
- 向WMS添加一个窗口。必须在LayoutParams中指明此窗口所隶属于的窗口令牌，否则在某些情况下添加操作会失败。在SampleWindow中，不设置令牌也可成功?完成添加操作，因为窗口的类型被设为TYPE_SYSTEM_ALERT，它是系统窗口的一种。而对于系统窗口，WMS会自动为其创建显示令牌，故无需客户端操心。此话题将会在后文进行更具体的讨论。
- 向WMS申请对窗口进行重新布局（relayout）。所谓的重新布局，就是根据窗口新的属性去调整其Surface相关的属性，或者重新创建一个Surface（例如窗口尺寸变化导致之前的Surface不满足要求）。向WMS添加一个窗口之后，其仅仅是将它在WMS中进行了注册而已。只有经过重新布局之后，窗口才拥有WMS为其分配的画布。有了画布，窗口之后就可以随时进行绘制工作了。

而窗口的绘制过程如下：

- 通过Surface.lock()函数获取可以在其上作画的Canvas实例。
- 使用Canvas实例进行作画。
- 通过Surface.unlockCanvasAndPost()函数提交绘制结果。

提示关于Surface的原理与使用方法，请参考《卷 I》第8章“深入理解Surface系统”。

这是对Surface作画的标准方法。在客户端也可以通过OpenGL进行作画，不过这超出了本书的讨论范围。另外，在SampleWindow例子中使用了Choreographer类进行了动画帧的安排。Choreographer意为编舞指导，是Jelly Bean新增的一个工具类。其用法与Handler的post()函数非Z且不会再显示新的窗口，则需要从WMS将之前添加的显示令牌一并删除。

3. 窗口的概念

在SampleWindow例子中，有一个名为mWindow（类型为IWindow）的变量。读者可能会理所当然地认为它就是窗口了。其实这种认识并不完全正确。IWindow继承自Binder，并且其Bn端位于应用程序一侧（在例子中IWindow的实现类MyWindow就继承自IWindow.Stub），于是其在WMS一侧只能作为一个回调，以及起到窗口Id的作用。

那么，窗口的本质是什么呢？

是进行绘制所使用的画布：Surface。

当一块Surface显示在屏幕上时，就是用户所看到的窗口了。客户端向WMS添加一个窗口的过程，其实就是WMS为其分配一块Surface的过程，一块块Surface在WMS的管理之下有序地排布在屏幕上，Android才得以呈现出多姿多彩的界面来。所以从这个意义上来讲，WindowManagerService被称之为SurfaceManagerService也说得通的。

于是，根据对Surface的操作类型可以将Android的显示系统分为三个层次，如图4-2所示。

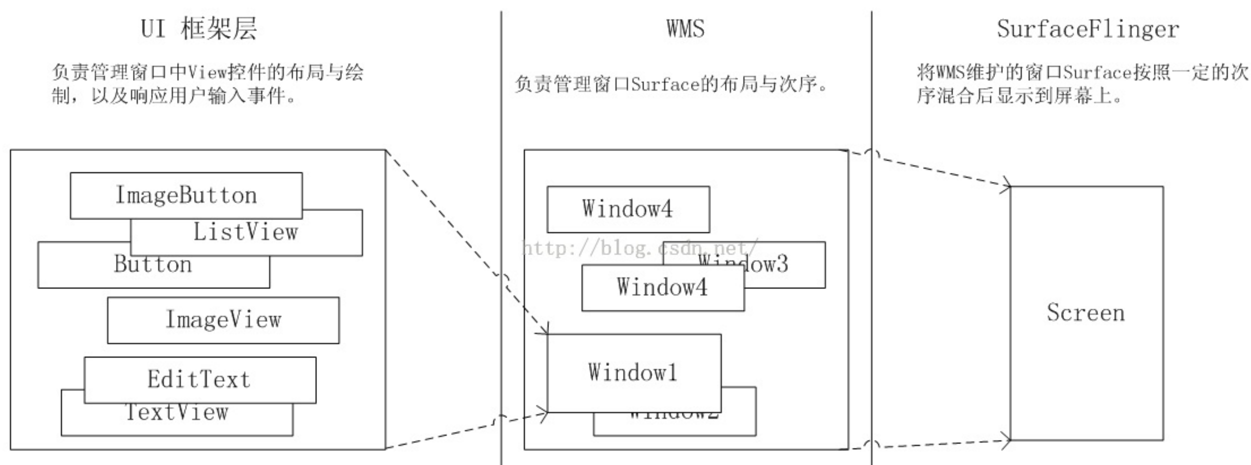


图 4-2 Android显示系统的三个层次

在图4-2中：

- 第一个层次是UI框架层，其工作为在Surface上绘制UI元素以及响应输入事件。
- 第二个层次为WMS，其主要工作在于管理Surface的分配、层级顺序等。
- 第三层为SurfaceFlinger，负责将多个Surface混合并输出。

经过这个例子的介绍，相信大家对WMS的功能有了一个初步的了解。接下来，我们要进入WMS的内部，通过其启动过程一窥它的构成。

4.1.2WMS的构成

俗话说，一个好汉三个帮！WMS的强大是由很多重要的成员互相协调工作而实现的。了解WMS的构成将会为我们深入探索WMS打下良好的基础，进而分析它的启动过程，这是再合适不过了。

1. WMS的诞生

和其他的系统服务一样，WMS的启动位于SystemServer.java中ServerThread类的run()函数内。

```
[-->SystemServer.java::ServerThread.run()]
Public void run() {
    ......
   WindowManagerService wm = null;
    ......
    try {
       ......
        // **①创建WMS实例**
       /* 通过WindowManagerService的静态函数main()创建WindowManagerService的实例。
           注意main()函数的两个参数wmHandler和uiHandler。这两个Handler分别运行于由
           ServerThread所创建的两个名为“WindowManager”和“UI”的两个HandlerThread中 */
        wm =WindowManagerService.main(context, power, display, inputManager,
        uiHandler,wmHandler,
                    factoryTest !=SystemServer.FACTORY_TEST_LOW_LEVEL,
                   !firstBoot, onlyCore);
        // 添加到ServiceManager中去
        ServiceManager.addService(Context.WINDOW_SERVICE,wm);
       ......
    catch(RuntimeException e) {
        ......
    }
    ......
    try {
        //**②初始化显示信息**
        wm.displayReady();
    } catch(Throwable e) {......}
    ......
    try {
        // ③通知WMS，系统的初始化工作完成
        wm.systemReady();
    } catch(Throwable e) {......}
    ......
}
```

由此可以看出，WMS的创建分为三个阶段：

- 创建WMS的实例。
- 初始化显示信息。
- 处理systemReady通知。

接下来，将通过以上三个阶段分析WMS从无到有的过程。

看一下WMS的main()函数的实现：

```
[-->WindowManagerService.java::WindowManagerSrevice.main()]
public static WindowManagerService main(finalContext context,
    finalPowerManagerService pm, final DisplayManagerService dm,
    finalInputManagerService im,
    finalHandler uiHandler, final Handler wmHandler,
    finalboolean haveInputMethods, final boolean showBootMsgs,
    finalboolean onlyCore) {
    finalWindowManagerService[] holder = new WindowManagerService[1];
    // 通过由SystemServer为WMS创建的Handler新建一个WindowManagerService对象
    // 此Handler运行在一个名为WindowManager的HandlerThread中
    wmHandler.runWithScissors(newRunnable() {
       @Override
        publicvoid run() {
            holder[0]= new WindowManagerService(context, pm, dm, im,
                            uiHandler,haveInputMethods, showBootMsgs, onlyCore);
        }
    }, 0);
    returnholder[0];
}
```

注意 Handler类在Android 4.2中新增了一个API：runWithScissors()。这个函数将会在Handler所在的线程中执行传入的Runnable对象，同时阻塞调用线程的执行，直到Runnable对象的run()函数执行完毕。

WindowManagerService.main()函数在ServerThread专为WMS创建的线程“WindowManager”上创建了一个WindowManagerService的新实例。WMS中所有需要的Looper对象，例如Handler、Choreographer等，将会运行在“WindowManager”线程中。

接下来看一下其构造函数，看一下WMS定义了哪些重要的组件。

```
[-->WindowManagerService.java::WindowManagerService.WindowManagerService()]
private WindowManagerService(Context context,PowerManagerService pm,
           DisplayManagerService displayManager, InputManagerService inputManager,
           Handler uiHandler,
    booleanhaveInputMethods, boolean showBootMsgs, boolean onlyCore)
    ......
    mDisplayManager=
             (DisplayManager)context.getSystemService(Context.DISPLAY_SERVICE);
    mDisplayManager.registerDisplayListener(this,null);
    Display[]displays = mDisplayManager.getDisplays();
    /* 初始化DisplayContent列表。DisplayContent是Android4.2为支持多屏幕输出所引入的一个
       概念。一个DisplayContent指代一块屏幕，屏幕可以是手机自身的屏幕，也可以是基于Wi-FiDisplay
       技术的虚拟屏幕[3]*/
    for(Display display : displays) {
        createDisplayContentLocked(display);
    }
    .....
    /* 保存InputManagerService。输入事件最终要分发给具有焦点的窗口，而WMS是窗口管理者，
       所以WMS是输入系统中的重要一环。关于输入系统的内容将在第5章中深入探讨*/
    mInputManager= inputManager;
    // 这个看起来其貌不扬的mAnimator，事实上具有非常重要的作用。它管理着所有窗口的动画
    mAnimator= new WindowAnimator(this, context, mPolicy);
    // 在“UI“线程中将对另一个重要成员mPolicy，也就是WindowManagerPolicy进行初始化
    initPolicy(uiHandler);
    // 将自己加入到Watchdog中
    Watchdog.getInstance().addMonitor(this);
    ......
}
```

第二步，displayReady()函数的调用主要是初始化显示尺寸的信息。其内容比较琐碎，这里就先不介绍了。不过值得注意的一点是，再displayReady()完成后，WMS会要求ActivityManagerService进行第一次Configuration的更新。

第三步，在systemReady()函数中，WMS本身将不会再做任何操作了，直接调用mPolicy的systemReady()函数。

2. WMS的重要成员

总结一下在WMS的启动过程中所创建的重要成员，参考图4-3。

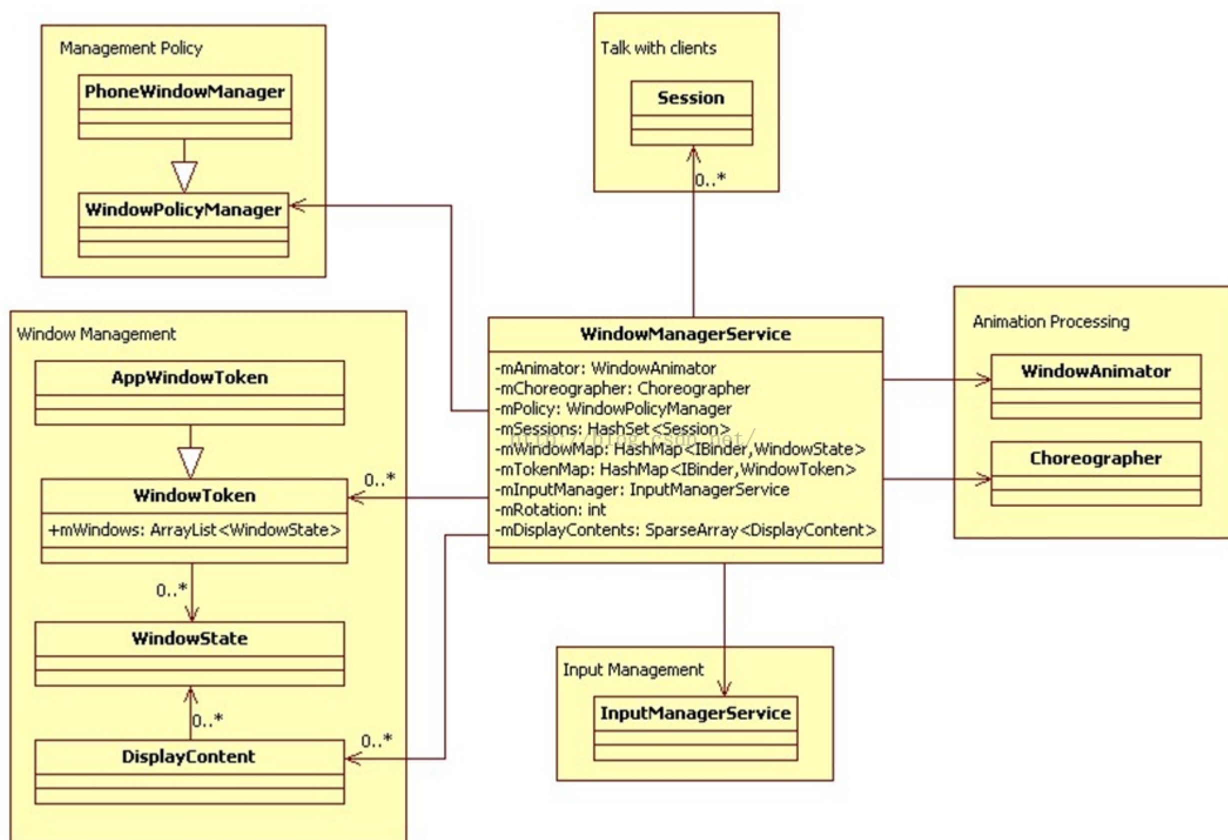


图 4-3 WMS的重要成员

以下是对图4-3中重要成员的简单介绍：

- mInputManager，InputManagerService（输入系统服务）的实例。用于管理每个窗口的输入事件通道（InputChannel）以及向通道上派发事件。关于输入系统的详细内容将在本书第5章详细探讨。
- mChoreographer，Choreographer的实例，在SampleWindow的例子中已经见过了。Choreographer的意思是编舞指导。它拥有从显示子系统获取VSYNC同步事件的能力，从而可以在合适的时机通知渲染动作，避免在渲染的过程中因为发生屏幕重绘而导致的画面撕裂。从这个意义上来讲，Choreographer的确是指导Android翩翩起舞的大师。WMS使用Choreographer负责驱动所有的窗口动画、屏幕旋转动画、墙纸动画的渲染。

- mAnimator，WindowAnimator的实例。它是所有窗口动画的总管（窗口动画是一个WindowStateAnimator的对象）。在Choreographer的驱动下，逐个渲染所有的动画。

- mPolicy，WindowPolicyManager的一个实现。目前它只有PhoneWindowManager一个实现类。mPolicy定义了很多窗口相关的策略，可以说是WMS的首席顾问！每当WMS要做什么事情的时候，都需要向这个顾问请教应当如何做。例如，告诉WMS某一个类型的Window的ZOrder的值是多少，帮助WMS矫正不合理的窗口属性，会为WMS监听屏幕旋转的状态，还会预处理一些系统按键事件（例如HOME、BACK键等的默认行为就是在这里实现的），等等。所以，mPolicy可谓是WMS中最重要的一个成员了。

- mDisplayContents，一个DisplayContent类型的列表。Android4.2支持基于Wi-fi Display的多屏幕输出，而一个DisplayContent描述了一块可以绘制窗口的屏幕。每个DisplayContent都用一个整型变量作为其ID，其中手机默认屏幕的ID由Display.DEFAULT_DISPLAY常量指定。DisplayContent的管理是由DisplayManagerService完成的，在本章不会去探讨DisplayContent的实现细节，而是关注DisplayContent对窗口管理与布局的影响。

下面的几个成员的初始化并没有出现在构造函数中，不过它们的重要性一点也不亚于上面几个。

- mTokenMap，一个HashMap，保存了所有的显示令牌（类型为WindowToken），用于窗口管理。在SampleWindow例子中曾经提到过，一个窗口必须隶属于某一个显示令牌。在那个例子中所添加的令牌就被放进了这个HashMap中。从这个成员中还衍生出几个辅助的显示令牌的子集，例如mAppTokens保存了所有属于Activity的显示令牌（WindowToken的子类AppWindowToken），mExitingTokens则保存了正在退出过程中的显示令牌等。其中mAppTokens列表是有序的，它与AMS中的mHistory列表的顺序保持一致，反映了系统中Activity的顺序。

- mWindowMap，也是一个HashMap，保存了所有窗口的状态信息（类型为WindowState），用于窗口管理。在SampleWindow例子中，使用IWindowSession.add()所添加的窗口的状态将会被保存在mWindowMap中。与mTokenMap一样，mWindowMap一样有衍生出的子集。例如mPendingRemove保存了那些退出动画播放完成并即将被移除的窗口，mLosingFocus则保存了那些失去了输入焦点的窗口。在DisplayContent中，也有一个windows列表，这个列表存储了显示在此DisplayContent中的窗口，并且它是有序的。窗口在这个列表中的位置决定了其最终显示时的Z序。

- mSessions，一个List，元素类型为Session。Session其实是SampleWindow例子中的IWindowSession的Bn端。也就是说，mSessions这个列表保存了当前所有想向WMS寻求窗口管理服务的客户端。注意Session是进程唯一的。

- mRotation，只是一个int型变量。它保存了当前手机的旋转状态。

WMS定义的成员一定不止这些，但是它们是WMS每一种功能最核心的变量。读者在这里可以线对它们有一个感性的认识。在本章后续的内容里将会详细分析它们在WMS的各种工作中所发挥的核心作用。

4.1.3 初识WMS的小结

这一节通过SampleWindow的例子向读者介绍了WMS的客户端如何使用窗口，然后通过WMS的诞生过程简单剖析了一下WMS的重要成员组成，以期通过本节的学习能够为后续的学习打下基础。

从下一节开始，我们将会深入探讨WMS的工作原理。

4.2 WMS的窗口管理结构

经过上一节的介绍，读者应该对WMS的窗口管理有了一个感性的认识。从这一节开将深入WMS的内部去剖析其工作流程。

根据前述内容可知，SampleWindow添加窗口的函数是IWindowSession.add()。IWindowSession是WMS与客户端交互的一个代理，add则直接调用到了WMS的addWindow()函数。我们将从这个函数开始WMS之旅。本小节只讨论它的前半部分。

注意 由于篇幅所限，本章不准备讨论removeWindow的实现。

```
[-->WindowManagerService.java::WindowManagerService.addWindow()Part1]
public int addWindow(Session session, IWindowclient, int seq,
           WindowManager.LayoutParams attrs, int viewVisibility,int displayId
           Rect outContentInsets, InputChannel outInputChannel) {
        // 首先检查权限，没有权限的客户端不能添加窗口
        intres = mPolicy.checkAddPermission(attrs);
       ......
        // 当为某个窗口添加子窗口时，attachedWindow将用来保存父窗口的实例
       WindowState attachedWindow = null;
        //win就是即将被添加的窗口了
       WindowState win = null;
       ......
        finalint type = attrs.type;
        synchronized(mWindowMap){
           ......
           //①获取窗口要添加到的DisplayContent
           /* 在添加窗口时，必须通过displayId参数指定添加到哪一个DisplayContent。
              SampleWindow例子没有指定displayId参数，Session会替SampleWindow选择
              DEFAULT_DISPLAY，也就是手机屏幕 */
            finalDisplayContent displayContent = getDisplayContentLocked(displayId);
            if(displayContent == null) {
              return WindowManagerGlobal.ADD_INVALID_DISPLAY;
           }
           ......
           // 如果要添加的窗口是另一个的子窗口，就要求父窗口必须已经存在            // 注意， attrs.
           // 对于子窗口来说，attrs.token表示了父窗口
            if(type &gt;= FIRST_SUB_WINDOW &&.type &lt;= LAST_SUB_WINDOW) {
              attachedWindow = windowForClientLocked(null, attrs.token, false);
              if (attachedWindow == null) {
                  return WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN;
              }
              //在这里还可以看出WMS要求窗口的层级关系最多为两层
              if (attachedWindow.mAttrs.type &gt;= FIRST_SUB_WINDOW
```

```
                        &&attachedWindow.mAttrs.type &lt;= LAST_SUB_WINDOW) {
                return WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN;
            }
        }
         booleanaddToken = false;
        // **②WindowToken出场！**根据客户端的attrs.token取出已注册的WindowToken
        WindowToken token = mTokenMap.get(attrs.token);
        // 下面的if语句块初步揭示了WindowToken和窗口之间的关系
         if(token == null) {
            // 对于以下几种类型的窗口，必须通过LayoutParams.token成员为其指定一个已经
            // 添加至WMS的WindowToken
            if (type &gt;= FIRST_APPLICATION_WINDOW
                        && type&lt;= LAST_APPLICATION_WINDOW) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
            if (type == TYPE_INPUT_METHOD) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
            if (type == TYPE_WALLPAPER) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
            if (type == TYPE_DREAM) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
            // 其他类型的窗口则不需要事先向WMS添加WindowToken因为WMS会在这里隐式地创
            // 建一个。注意最后一个参数false，这表示此WindowToken由WMS隐式创建。
            token = new WindowToken(this, attrs.token, -1, false);
             addToken = true;
        } else if (type &gt;= FIRST_APPLICATION_WINDOW
                                    &&type &lt;= LAST_APPLICATION_WINDOW) {
            // 对于APPLICATION类型的窗口，要求对应的WindowToken的类型也为APPLICATION
            // 并且是WindowToken的子类：AppWindowToken
            AppWindowToken atoken = token.appWindowToken;
            if (atoken == null) {
                return WindowManagerImpl.ADD_NOT_APP_TOKEN;
            } else if (atoken.removed) {
                 returnWindowManagerImpl.ADD_APP_EXITING;
            }
            if (type==TYPE_APPLICATION_STARTING && atoken.firstWindowDrawn){
                return WindowManagerImpl.ADD_STARTING_NOT_NEEDED;
            }
        } else if (type == TYPE_INPUT_METHOD) {
            // 对于其他几种类型的窗口也有类似的要求：窗口类型必须与WindowToken的类型一致
            if (token.windowType != TYPE_INPUT_METHOD) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
        } else if (type == TYPE_WALLPAPER) {
            if (token.windowType != TYPE_WALLPAPER) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
        } else if (type == TYPE_DREAM) {
            if (token.windowType != TYPE_DREAM) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
        }
        // ③WMS为要添加的窗口创建了一个WindowState对象
        // 这个对象维护了一个窗口的所有状态信息
         win= new WindowState(this, session, client, token,
         attachedWindow,seq, attrs, viewVisibility, displayContent);
        ......
        // WindowManagerPolicy出场了。这个函数的调用会调整LayoutParams的一些成员的取值
         mPolicy.adjustWindowParamsLw(win.mAttrs);
         res= mPolicy.prepareAddWindowLw(win, attrs);
         if(res != WindowManagerGlobal.ADD_OKAY) {
            return res;
        }
        // 接下来将刚刚隐式创建的WindowToken添加到mTokenMap中去。通过这行代码应该
        //读者应该能想到，所有的WindowToken都被放入这个HashTable中
         ......
         if(addToken) {
            mTokenMap.put(attrs.token, token);
        }
```

```
            win.attach();
           // 然后将WindowState对象加入到mWindowMap中
            mWindowMap.put(client.asBinder(),win);
           // 剩下的代码稍后再做分析
           ......
        }
    }
```

addWindow()函数的前段代码展示了三个重要的概念，分别是WindowToken、WindowState以及DisplayContent。并且在函数开始处对窗口类型的检查判断也初步揭示了它们之间的关系：除子窗口外，添加任何一个窗口都必须指明其所属的WindowToken；窗口在WMS中通过一个WindowState实例进行管理和保管。同时必须在窗口中指明其所属的DisplayContent，以便确定窗口将被显示到哪一个屏幕上。

4.2.1 理解WindowToken

1. WindowToken的意义

为了搞清楚WindowToken的作用是什么，看一下其位于WindowToken.java中的定义。虽然它没有定义任何函数，但其成员变量的意义却很重要。

- WindowToken将属于同一个应用组件的窗口组织在了一起。所谓的应用组件可以是Activity、InputMethod、Wallpaper以及Dream。在WMS对窗口的管理过程中，用WindowToken指代一个应用组件。例如在进行窗口ZOrder排序时，属于同一个WindowToken的窗口会被安排在一起，而且在其中定义的一些属性将会影响所有属于此WindowToken的窗口。这些都表明了属于同一个WindowToken的窗口之间的紧密联系。
- WindowToken具有令牌的作用，是对应用组件的行为进行规范管理的一个手段。WindowToken由应用组件或其管理者负责向WMS声明并持有。应用组件在需要新的窗口时，必须提供WindowToken以表明自己的身份，并且窗口的类型必须与所持有的WindowToken的类型一致。从上面的代码可以看到，在创建系统类型的窗口时不需要提供一个有效的Token，WMS会隐式地为其声明一个WindowToken，看起来谁都可以添加个系统级的窗口。难道Android为了内部使用方便而置安全于不顾吗？非也，addWindow()函数一开始的mPolicy.checkAddPermission()的目的就是如此。它要求客户端必须拥有SYSTEM_ALERT_WINDOW或INTERNAL_SYSTEM_WINDOW权限才能创建系统类型的窗口。

2. 向WMS声明WindowToken

既然应用组件在创建一个窗口时必须指定一个有效的WindowToken才行，那么WindowToken究竟该如何声明呢？

在SampleWindow应用中，使用wms.addWindowToken()函数声明mToken作为它的令牌，所以在添加窗口时，通过设置lp.token为mToken向WMS进行出示，从而获得WMS添加窗口的许可。这说明，只要是一个Binder对象（随便一个），都可以作为Token向WMS进行声明。对

于WMS的客户端来说，**Token**仅仅是一个**Binder**对象而已。

为了验证这一点，来看一下addWindowToken的代码，如下所示：

```
[-->WindowManagerService.java::WindowManagerService.addWindowToken()]
   @Override
    publicvoid addWindowToken(IBinder token, int type) {
        // 需要声明Token的调用者拥有MANAGE_APP_TOKENS的权限
        if(!checkCallingPermission(android.Manifest.permission.MANAGE_APP_TOKENS,
               "addWindowToken()")) {
            thrownew SecurityException("Requires MANAGE_APP_TOKENS permission");
        }
        synchronized(mWindowMap){
            ......
           // 注意其构造函数的参数与addWindow()中不同，最后一个参数为true，表明这个Token
           // 是显式申明的
            wtoken= new WindowToken(this, token, type, true);
            mTokenMap.put(token,wtoken);
            ......
        }
    }
```

使用addWindowToken()函数声明Token，将会在WMS中创建一个WindowToken实例，并添加到mTokenMap中，键值为客户端用于声明Token的Binder实例。与addWindow()函数中隐式地创建WindowToken不同，这里的WindowToken被声明为显式的。隐式与显式的区别在于，当隐式创建的WindowToken的最后一个窗口被移除后，此WindowToken会被一并从mTokenMap中移除。显式创建的WindowToken只能通过removeWindowToken()显式地移除。

addWindowToken()这个函数告诉我们，WindowToken其实有两层含义：

- 对于显示组件（客户端）而言的Token，是任意一个Binder的实例，对显示组件（客户端）来说仅仅是一个创建窗口的令牌，没有其他的含义。
- 对于WMS而言的WindowToken这是一个WindowToken类的实例，保存了对应于客户端一侧的Token（Binder实例），并以这个Token为键，存储于mTokenMap中。客户端一侧的Token是否已被声明，取决于其对应的WindowToken是否位于mTokenMap中。

注意 在一般情况下，称显示组件（客户端）一侧Binder的实例为Token，而称WMS一侧的WindowToken对象为WindowToken。但是为了叙述方便，在没有歧义的前提下不会过分仔细地区分这两个概念。

接下来，看一下各种显示组件是如何声明WindowToken的。

（1） Wallpaper和InputMethod的Token

Wallpaper的Token声明在WallpaperManagerService中。参考以下代码：

```
[-->WallpaperManagerService.java::WallpaperManagerService.bindWallpaperComponentLocked()]
BooleanbindWallpaperComponentLocked(......) {
    ......
   WallpaperConnection newConn = new WallpaperConnection(wi, wallpaper);
    ......
    mIWindowManager.addWindowToken(newConn.mToken,
                         WindowManager.LayoutParams.TYPE_WALLPAPER);
    ......
}
```

WallpaperManagerService是Wallpaper管理器，它负责维护系统已安装的所有的Wallpaper并在它们之间进行切换，而这个函数的目的是准备显示一个Wallpaper。newConn.mToken与SampleWindow例子一样，是一个简单的Binder对象。这个Token将在即将显示出来的Wallpaper被连接时传递给它，之后Wallpaper即可通过这个Token向WMS申请创建绘制壁纸所需的窗口了。

注意 WallpaperManagerService向WMS声明的Token类型为TYPE_WALLPAPER，所以，Wallpaper仅能本分地创建TYPE_WALLPAPER类型的窗口。

相应的，WallpaperManagerService会在detachWallpaperLocked()函数中取消对Token的声明：

```
[-->WallpaperManagerService.java::WallpaperManagerService.detachWallpaperLocked()]
booleandetachWallpaperLocked(WallpaperData wallpaper){
    ......
    mIWindowManager.removeWindowToken(wallpaper.connection.mToken);
    ......
}
```

再此之后，如果这个被detach的Wallpaper想再要创建窗口便不再可能了。

WallpaperManagerService使用WindowToken对一个特定的Wallpaper做出了如下限制：

- Wallpaper只能创建TYPE_WALLPAPER类型的窗口。
- Wallpaper显示的生命周期由WallpaperManagerService牢牢地控制着。仅有当前的Wallpaper才能创建窗口并显示内容。其他的Wallpaper由于没有有效的Token，而无法创建窗口。

InputMethod的Token的来源与Wallpaper类似，其声明位于InputMethodManagerService的startInputInnerLocked()函数中，取消声明的位置在InputmethodManagerService的unbindCurrentMethodLocked()函数。InputMethodManagerService通过Token限制着每一个InputMethod的窗口类型以及显示生命周期。

（2） Activity的Token

Activity的Token的使用方式与Wallpaper和InputMethod类似，但是其包含更多的内容。毕竟，对于Activity，无论是其组成还是操作都比Wallpaper以及InputMethod复杂得多。对此，WMS专为Activity实现了一个WindowToken的子类：AppWindowToken。

既然AppWindowToken是为Activity服务的，那么其声明自然在ActivityManagerService中。具体位置为ActivityStack.startActivityLocked()，也就是启动Activity的时候。相关代码如下：

```
[-->ActivityStack.java::ActivityStack.startActivityLocked()]
private final void startActivityLocked(......) {
    ......
    mService.mWindowManager.addAppToken(addPos,r.appToken, r.task.taskId,
                               r.info.screenOrientation, r.fullscreen);
    ......
}
```

startActivityLocked()向WMS声明r.appToken作为此Activity的Token，这个Token是在ActivityRecord的构造函数中创建的。随然后在realStartActivityLocked()中将此Token交付给即将启动的Activity。

```
[-->ActivityStack.java::ActivityStack.realStartActivityLocked()]
final boolean realStartActivityLocked(......) {
    ......
    app.thread.scheduleLaunchActivity(newIntent(r.intent), **r.appToken,**
                System.identityHashCode(r), r.info,
                newConfiguration(mService.mConfiguration),
                r.compat, r.icicle, results, newIntents,!andResume,
    mService.isNextTransitionForward(),profileFile, profileFd,
                profileAutoStop);
    ......
}
```

启动后的Activity即可使用此Token创建类型为TYPE_APPLICATION的窗口了。

取消Token的声明则位于ActivityStack.removeActivityFromHistoryLocked()函数中。

Activity的Token在客户端是否和Wallpaper一样，仅仅是一个基本的Binder实例呢？其实不然。看一下r.appToken的定义可以发现，这个Token的类型是IApplicationToken.Stub。其中定义了一系列和窗口相关的一些通知回调，它们是：

- windowsDrawn()，当窗口完成初次绘制后通知AMS。
- windowsVisible()，当窗口可见时通知AMS。
- windowsGone()，当窗口不可见时通知AMS。
- keyDispatchingTimeout()，窗口没能按时完成输入事件的处理。这个回调将会导致ANR。
- getKeyDispatchingTimeout()，从AMS处获取界定ANR的时间。

AMS通过ActivityRecord表示一个Activity。而ActivityRecord的appToken在其构造函数中被创建，所以每个ActivityRecord拥有其各自的appToken。而WMS接受AMS对Token的声明，并为appToken创建了唯一的一个AppWindowToken。因此，这个类型为IApplicationToken的Binder对象appToken粘结了AMS的ActivityRecord与WMS的AppWindowToken，只要给定一个ActivityRecord，都可以通过appToken在WMS中找到一个对应的AppWindowToken，从而使得AMS拥有了操纵Activity的窗口绘制的能力。例如，当AMS认为一个Activity需要被隐藏时，以Activity对应的ActivityRecord所拥有的appToken作为参数调用WMS的setAppVisibility()函数。此函数通过appToken找到其对应的AppWindowToken，然后将属于这个Token的所有窗口隐藏。

注意 每当AMS因为某些原因（如启动/结束一个Activity，或将Task移到前台或后台）而调整ActivityRecord在mHistory中的顺序时，都会调用WMS相关的接口移动AppWindowToken在mAppTokens中的顺序，以保证两者的顺序一致。在后面讲解窗口排序规则时会介绍到，AppWindowToken的顺序对窗口的顺序影响非常大。

4.2.2 理解WindowState

从WindowManagerService.addWindow()函数的实现中可以看出，当向WMS添加一个窗口时，WMS会为其创建一个WindowState。WindowState表示一个窗口的所有属性，所以它是WMS中事实上的窗口。这些属性将在后面遇到时再做介绍。

类似于WindowToken，WindowState在显示组件一侧也有个对应的类型：IWindow.Stub。IWindow.Stub提供了很多与窗口管理相关通知的回调，例如尺寸变化、焦点变化等。

另外，从WindowManagerService.addWindow()函数中看到新的WindowState被保存到mWindowMap中，键值为IWindow的Bp端。mWindowMap是整个系统所有窗口的一个全集。

说明对比一下mTokenMap和mWindowMap。这两个HashMap维护了WMS中最重要的两类数据：WindowToken及WindowState。它们的键都是IBinder，区别是： mTokenMap的键值可能是IAppWindowToken的Bp端（使用addAppToken()进行声明），或者是其他任意一个Binder的Bp端(使用addWindowToken()进行声明)；而mWindowToken的键值一定是IWindow的Bp端。

关于WindowState的更多细节将在后面的讲述中进行介绍。不过经过上面的分析，不难得到WindowToken和WindowState之间的关系，参考图4-4。

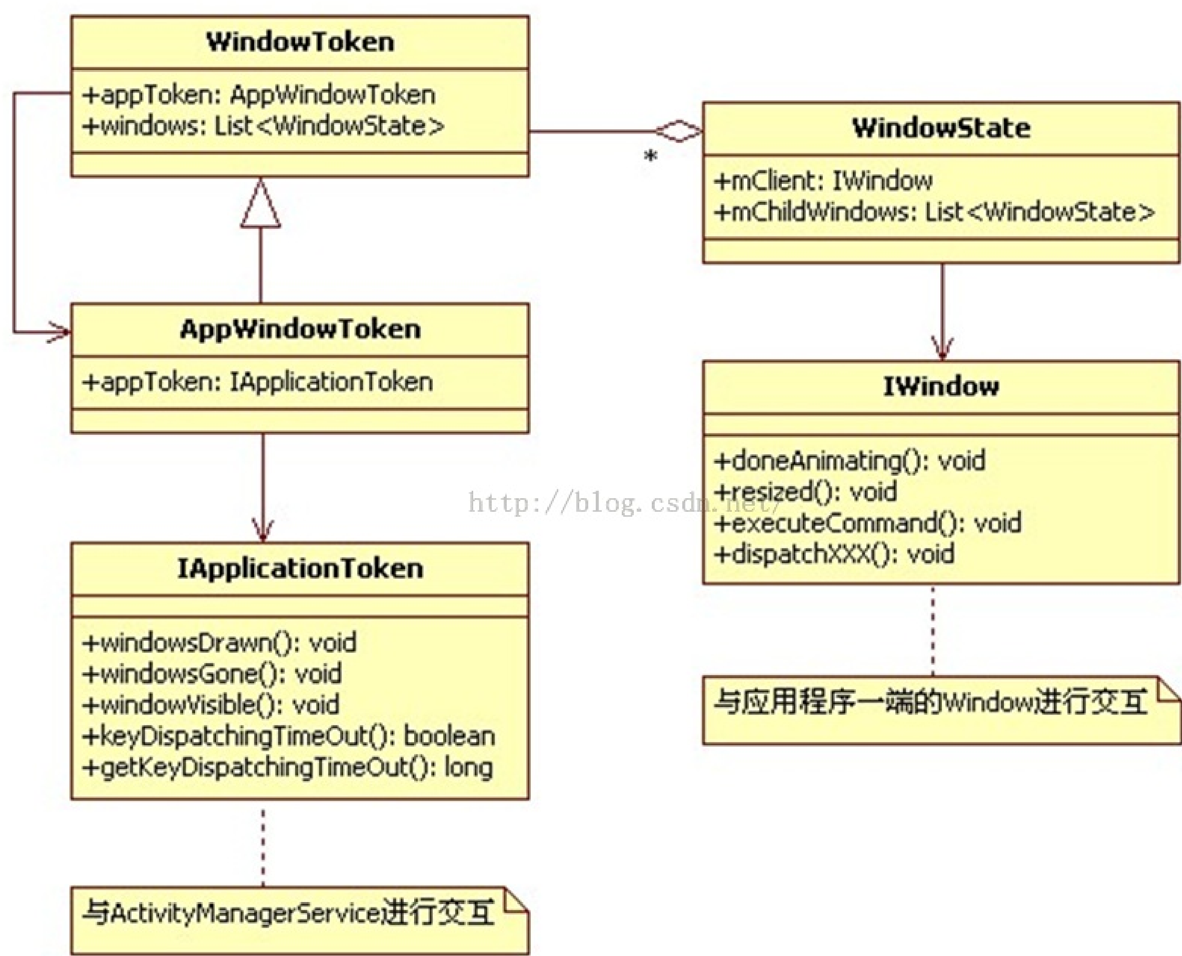


图 4-4 WindowToken与WindowState的关系

更具体一些，以一个正在回放视频并弹出两个对话框的Activity为例，WindowToken与WindowState的意义如图4-5所示。

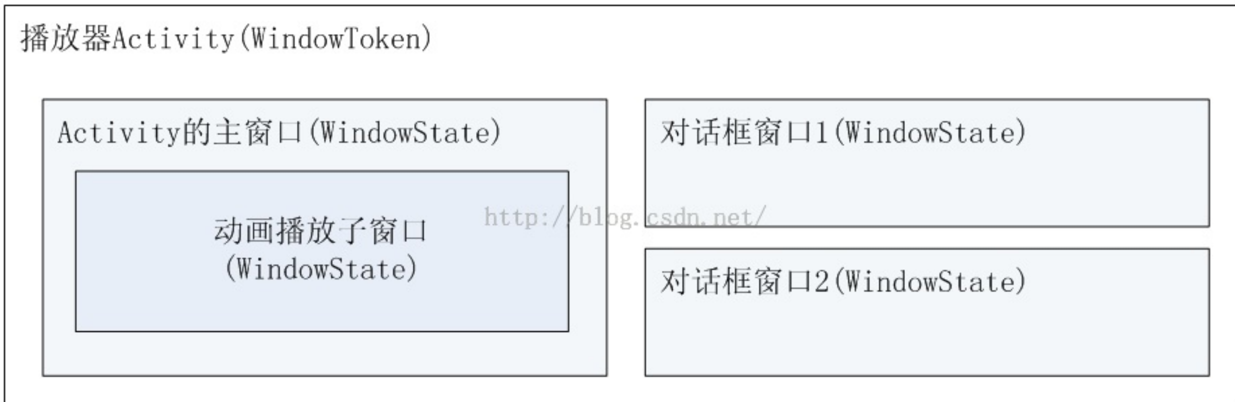


图 4-5WindowState与WindowToken的从属关系

4.2.3理解DisplayContent

如果说WindowToken按照窗口之间的逻辑关系将其分组，那么DisplayContent则根据窗口的显示位置将其分组。隶属于同一个DisplayContent的窗口将会被显示在同一个屏幕中。每一个DisplayContent都对应这一个唯一的ID，在添加窗口时可以通过指定这个ID决定其将被显示在那个屏幕中。

DisplayContent是一个非常具有隔离性的一个概念。处于不同DisplayContent的两个窗口在布局、显示顺序以及动画处理上不会产生任何耦合。因此，就这几个方面来说，DisplayContent就像一个孤岛，所有这些操作都可以在其内部独立执行。因此，这些本来属于整个WMS全局性的操作，变成了DisplayContent内部的操作了。

4.3 理解窗口的显示次序

在addWindow()函数的前半部分中，WMS为窗口创建了用于描述窗口状态的WindowState，接下来便会为新建的窗口确定显示次序。手机屏幕是以左上角为原点，向右为X轴方向，向下为Y轴方向的一个二维空间。为了方便管理窗口的显示次序，手机的屏幕被扩展为了一个三维的空间，即多定义了一个Z轴，其方向为垂直于屏幕表面指向屏幕外。多个窗口依照其前后顺序排布在这个虚拟的Z轴上，因此窗口的显示次序又被称为Z序（Z order）。在这一节中将深入探讨WMS确定窗口显示次序的过程以及其影响因素。

4.3.1 主序、子序和窗口类型

看一下WindowState的构造函数：

```
[-->WindowState.java::WindowState.WindowState()]
WindowState(WindowManagerService service, Sessions, IWindow c, WindowToken token,
      WindowState attachedWindow, int seq, WindowManager.LayoutParams a,
    intviewVisibility, final DisplayContent displayContent) {
    ......
    // 为子窗口分配ZOrder
    if((mAttrs.type &gt;= FIRST_SUB_WINDOW &&
           mAttrs.type &lt;= LAST_SUB_WINDOW)) {
        // 这里的mPolicy即是WindowManagerPolicy
        mBaseLayer= mPolicy.windowTypeToLayerLw(
               attachedWindow.mAttrs.type)
               * WindowManagerService.TYPE_LAYER_MULTIPLIER
               + WindowManagerService.TYPE_LAYER_OFFSET;
        mSubLayer= mPolicy.subWindowTypeToLayerLw(a.type);
       ......
    } else {// 为普通窗口分配ZOrder
        mBaseLayer= mPolicy.windowTypeToLayerLw(a.type)
               * WindowManagerService.TYPE_LAYER_MULTIPLIER
               + WindowManagerService.TYPE_LAYER_OFFSET;
        mSubLayer= 0;
       ......
    }
    ......
}
```

窗口的显示次序由两个成员字段描述：主序mBaseLayer和子序mSubLayer。主序用于描述窗口及其子窗口在所有窗口中的显示位置。而子序则描述了一个子窗口在其兄弟窗口中的显示位置。

- 主序越大，则窗口及其子窗口的显示位置相对于其他窗口的位置越靠前。
- 子序越大，则子窗口相对于其兄弟窗口的位置越靠前。对于父窗口而言，其主序取决于其类型，其子序则保持为0。而子窗口的主序与其父窗口一样，子序则取决于其类型。从上述代码可以看到，主序与子序的分配工作是由WindowManagerPolicy的两个成员函数windowTypeToLayerLw()和subWindowTypeToLayerLw()完成的。

表4-1与表4-2列出了所有可能的窗口类型以及其主序与子序的值。

表 4-1 窗口的主序

窗口类型	主序	窗口类型	
TYPE_UNIVERSE_BACKGROUND	11000	TYPE_WALLPAPER	2
TYPE_PHONE	31000	TYPE_SEARCH_BAR	4
TYPE_RECENTS_OVERLAY	51000	TYPE_SYSTEM_DIALOG	5
TYPE_TOAST	61000	TYPE_PRIORITY_PHONE	7
TYPE_DREAM	81000	TYPE_SYSTEM_ALERT	9
TYPE_INPUT_METHOD	101000	TYPE_INPUT_METHOD_DIALOG	1
TYPE_KEYGUARD	121000	TYPE_KEYGUARD_DIALOG	1
TYPE_STATUS_BAR_SUB_PANEL	141000	应用窗口与未知类型的窗口	2

表 4-2 窗口的子序

子窗口类型	子序
TYPE_APPLICATION_PANEL	1
TYPE_APPLICATION_ATTACHED_DIALOG	1
TYPE_APPLICATION_MEDIA	-2
TYPE_APPLICATION_MEDIA_OVERLAY	-1
TYPE_APPLICATION_SUB_PANEL	2

注意 表4-2中的MEDIA和MEDIA_OVERLAY的子序为负值，这表明它们的显示次序位于其父窗口的后面。这两个类型的子窗口是SurfaceView控件创建的。SurfaceView被实例化后，会向WMS添加一个类型为MEDIA的子窗口，它的父窗口就是承载SurfaceView控件的窗口。这个子窗口的Surface将被用于视频回放、相机预览或游戏绘制。为了不让这个子窗口覆盖住所有的父窗口中承载的其他控件（如拍照按钮，播放器控制按钮等），它必须位于父窗口之后。

从表4-1所描述的主序与窗口类型的对应关系中可以看出，WALLPAPER类型的窗口的主序竟和APPLICATION类型的窗口主序相同，这看似有点不合常理，WALLPAPER不是应该显示在所有Acitivity之下吗？其实WALLPAPER类型的窗口是一个很不安分的角色，需要在所有的APPLICATION窗口之间跳来跳去。这是因为，有的Activity指定了android:windowShowWallpaper为true，则表示窗口要求将用户当前壁纸作为其背景。对于WMS来说，最简单的办法就是将WALLPAPER窗口放置到紧邻拥有这个式样的窗口的下方。在这种需求下，为了保证主序决定窗口顺序的原则，WALLPAPER使用了与APPLICATION相同的主序。另外，输入法窗口也是一个很特殊的情况，输入法窗口会选择输入目标窗口，并将自己放置于其上。在本章中不讨论这两个特殊的例子，WALLPAPER的排序规则将在第7章中进行介绍，而输入法的排序则留给读者自行研究。

虽然知道了窗口的主序与子序是如何分配的，不过我们仍然存有疑问：如果有两个相同类型的窗口，那么它们的主序与子序岂不是完全相同？如何确定它们的显示顺序呢？事实上，表4-1和表4-2中所描述的主序和子序仅仅是排序的依据之一，WMS需要根据当前所有同类型窗口的数量为每个窗口计算最终的现实次序。

4.3.2 通过主序与子序确定窗口的次序

回到WMS的addWindow()函数中，继续往下看：

```
[-->WindowManagerService.java::WindowManagerService.addWindow()]
public int addWindow(Session session, IWindowclient, int seq,
   WindowManager.LayoutParams attrs, int viewVisibility, int displayId,
    RectoutContentInsets, InputChannel outInputChannel) {
    ......
    synchronized(mWindowMap){
        //在前面的代码中，WMS验证了添加窗口的令牌的有效性，并为新窗口创建了新的WindowState对象
        // 新的WindowState对象在其构造函数中根据窗口类型初始化了其主序mBaseLayer和mSubLayer
       ......
        // 接下来，将新的WindowState按照显示次序插入到当前DisplayContent的mWindows列表中
        // 为了代码结构的清晰，不考虑输入法窗口和壁纸窗口的处理
    if (type== TYPE_INPUT_METHOD) {
        ......
        }else if (type == TYPE_INPUT_METHOD_DIALOG) {
        }else {
           // 将新的WindowState按显示次序插入到当前DisplayContent的mWindows列表中
            addWindowToListInOrderLocked(win,true);
            if(type == TYPE_WALLPAPER) {
               ......
           }
        }
       ......
        // 根据窗口的排序结果，为DisplayContent的所有窗口分配最终的显示次序
        assignLayersLocked(displayContent.getWindowList());
        ......
    }
    ......
    returnres;
}
```

这里有两个关键点：

- addWindowToListInOrderLocked()将新建的WindowState按照一定的顺序插入到当前DisplayContent的mWindows列表中。在分析WMS的重要成员时提到过这个列表。它严格地按照显示顺序存储了所有窗口的WindowState。
- assignLayersLocked()将根据mWindows的存储顺序对所有的WindowState的主序和子序进行调整。

接下来分别分析一下这两个函数。

1. addWindowToListInOrderLocked()分析

addWindowToListInOrderLocked()的代码很长，不过其排序原则却比较清晰。这里直接给出其处理原则，感兴趣的读者可根据这些原则自行深究相关代码。

注意 再次强调一下，mWindows列表是按照主序与子序的升序进行排序的，所以显示靠前的窗口放在列表靠后的位置，而显示靠前的窗口，则位于列表的前面。也就是说，列表顺序与显示顺序是相反的。这点在阅读代码时要牢记，以免混淆。

在后面的叙述中，非特别强调，所谓的前后都是指显示顺序而不是在列表的存储顺序。

子窗口的排序规则：子窗口的位置计算是相对父窗口的，并根据其子序进行排序。由于父窗口的子序为0，所以子序为负数的窗口会放置在父窗口的后面，而子序为正数的窗口会放置在父窗口的前面。如果新窗口与现有窗口子序相等，则正数子序的新窗口位于现有窗口的前面，负数子序的新窗口位于现有窗口的后面。

非子窗口的排序则是依据主序进行的，但是其规则较为复杂，分为应用窗口和非应用窗口两种情况。之所以要区别处理应用窗口是因为所有的应用窗口的初始主序都是21000，并且应用窗口的位置应该与它所属的应用的其他窗口放在一起。例如应用A显示于应用B的后方，当应用A因为某个动作打开一个新的窗口时，新窗口应该位于应用A其他窗口的前面，但是不得覆盖应用B的窗口。只依据主序进行排序是无法实现这个管理逻辑的，还需要依赖Activity的顺序。在WindowToken一节的讲解中，曾经简单分析了mAppTokens列表的性质，它所保存的AppWindowToken的顺序与AMS中ActivityRecord的顺序时刻保持一致。因此，AppWindowToken在mAppTokens的顺序就是Activity的顺序。

非应用窗口的排序规则：依照主序进行排序，主序高者排在前面，当现有窗口的主序与新窗口相同时，新窗口位于现有窗口的前面。

应用窗口的排序规则：如上所述，同一个应用的窗口的显示位置必须相邻。如果当前应用已有窗口在显示（当前应用的窗口存储在其WindowState.appWindowToken.windows中），新窗口将插入到其所属应用其他窗口的前面，但是保证STARTING_WINDOW永远位于最前方，BASE_APPLICATION永远位于最后方。如果新窗口是当前应用的第一个窗口，则参照其他应用的窗口顺序，将新窗口插入到位于前面的最后一个应用的最后一个窗口的后方，或者位于后面的第一个应用的最前一个窗口的前方。如果当前没有其他应用的窗口可以参照，则直接根据主序将新窗口插入到列表中。

窗口排序的总结如下：

- 子窗口依据子序相对于其父窗口进行排序。相同子序的窗体，正子序则越新越靠前，负子序则越新越靠后。
- 应用窗口参照本应用其他窗口或相邻应用的窗口进行排序。如果没有任何窗口可以参照，则根据主序进行排序。
- 非应用窗口根据主序进行排序。

经过addWindowToListInOrderLocked()函数的处理之后，当前DisplayContent的窗口列表被插入了一个新的窗口。然后等待assignLayersLocked()的进一步处理。

2. assignLayersLocked分析

assignLayersLocked()函数将根据每个窗口的主序以及它们在窗口列表中的位置重新计算最终的显示次序mLayer。

```
[-->WindowManagerService.java::WindowManagerService.assignLayersLocked()]
privatefinal void assignLayersLocked(WindowList windows) {
    int N = windows.size();
    int curBaseLayer = 0;
    // curLayer表示当前分配到的Layer序号
    int curLayer = 0;
    int i;
    // 遍历列表中的所有的窗口，逐个分配显示次序
    for (i=0; i&lt;N; i++) {
        final WindowState w = windows.get(i);
        final WindowStateAnimator winAnimator =w.mWinAnimator;
        boolean layerChanged = false;
        int oldLayer = w.mLayer;
        if (w.mBaseLayer == curBaseLayer ||w.mIsImWindow
                || (i &gt; 0 &&w.mIsWallpaper)) {
            // 为具有相同主序的窗口在curLayer上增加一个偏移量，并将curLayer作为最终的显示次序
            curLayer +=WINDOW_LAYER_MULTIPLIER;
            w.mLayer = curLayer;
        } else {
            // 此窗口拥有不同的主序，直接将主序作为其显示次序并更新curLayer
            curBaseLayer = curLayer =w.mBaseLayer;
            w.mLayer = curLayer;
        }
        // 如果现实次序发生了变化则进行标记
        if (w.mLayer != oldLayer) {
            layerChanged = true;
            anyLayerChanged = true;
        }
        ......
    }
    ......
    // 向当前DisplayContent的监听者通知显示次序的更新
    if (anyLayerChanged) {
        scheduleNotifyWindowLayersChangedIfNeededLocked(
        getDefaultDisplayContentLocked());
    }
}
```

assignLayersLocked()的工作原理比较绕，简单来说，如果某个窗口在整个列表中拥有唯一的主序，则该主序就是其最终的显示次序。如果若干个窗口拥有相同的主序（注意经过addWindowToListInOrderLocked()函数的处理后，拥有相同主序的窗口都是相邻的），则第i

个相同主序的窗口的显示次序为在主序的基础上增加i * WINDOW_LAYER_MULTIPLIER的偏移。

经过assignLayersLocked()之后，一个拥有9个窗口的系统的现实次序的信息如表4-3所示。

表4- 3 窗口最终的显示次序信息

	窗口1	窗口2	窗口3	窗口4	窗口5	窗口6	窗口7	窗口
主序 mBaseLayer	11000	11000	21000	21000	21000	21000	71000	710
子序 mSubLayer	0	0	0	-1	0	0	0	0
显示次序 mLayer	11000	11005	21000	21005	21010	21015	71000	710

在确定了最终的显示次序mLayer后，又计算了WindowStateAnimator另一个属性：mAnimLayer。如下所示：

```
[-->WindowManagerService.java::assignLayersLocked()]
    finalWindowStateAnimator winAnimator = w.mWinAnimator;
    ......
    if (w.mTargetAppToken != null) {
        // 输入目标为Activity的输入法窗口，其mTargetAppToken是其输入目标所属的AppToken
       winAnimator.mAnimLayer =
               w.mLayer + w.mTargetAppToken.mAppAnimator.animLayerAdjustment;
    } elseif (w.mAppToken != null) {
        // 属于一个Activity的窗口
       winAnimator.mAnimLayer =
               w.mLayer + w.mAppToken.mAppAnimator.animLayerAdjustment;
    } else {
       winAnimator.mAnimLayer = w.mLayer;
    }
    ......
```

对于绝大多数窗口而言，其对应的WindowStateAnimator的mAnimLayer就是mLayer。而当窗口附属为一个Activity时，mAnimLayer会加入一个来自AppWindowAnimator的矫正：animLayerAdjustment。

WindowStateAnimator和AppWindowAnimator是动画系统中的两员大将，它们负责渲染窗口动画以及最终的Surface显示次序的修改。回顾一下4.1.2中的WMS的组成结构图，WindowState属于窗口管理体系的类，因此其所保存的mLayer的意义偏向于窗口管理。WindowStateAnimator/AppWindowAnimator则是动画体系的类，其mAnimLayer的意义偏向于动画，而且由于动画系统维护着窗口的Surface，因此**mAnimLayer是Surface**的实际显示次序。

在没有动画的情况下，mAnimLayer与mLayer是相等的，而当窗口附属为一个Activity时，则会根据AppTokenAnimator的需要适当地增加一个矫正值。这个矫正值来自AppTokenAnimator所使用的Animation。当Animation要求动画对象的ZOrder必须位于其他对象之上时（Animation.getZAdjustment()的返回值为Animation.ZORDER_TOP），这个矫正是一个正数

WindowManagerService.TYPE_LAYER_OFFSET（1000），这个矫正值很大，于是窗口在动画过程中会显示在其他同主序的窗口之上。相反，如果要求ZOrder必须位于其他对象之下时，矫正为-WindowManagerService.TYPE_LAYER_OFFSET(-1000)，于是窗口会显示在其他同主序的窗口之下。在动画完结后，mAnimLayer会被重新赋值为WindowState.mLayer，使得窗口回到其应有的位置。

动画系统的工作原理将在4.5节详细探讨。

注意 矫正值为常数1000，也就出现一个隐藏的bug：当同主序的窗口的数量大于200时，APPLICATION窗口的mLayer值可能超过22000。此时，在对于mLayer值为21000的窗口应用矫正后，仍然无法保证动画窗口位于同主序的窗口之上。不过超过200个应用窗口的情况非常少见，而且仅在动画过程中才会出现bug，所以google貌似也懒得解决这个问题。

4.3.3 更新显示次序到Surface

再回到WMS的addWindow()函数中，发现再没有可能和显示次序相关的代码了。mAnimLayer是如何发挥自己的作用呢？不要着急，事实上，新建的窗口目前尚无Surface。回顾一下SimpleWindow例子，在执行session.relayout()后，WMS才为新窗口分配了一块Surface。也就是说，只有执行relayout（）之后才会为新窗口的Surface设置新的显示次序。

为了不中断对显示次序的调查进展，就直接开门见山地告诉大家，设置显示次序到Surface的代码位于WindowStateAnimator. prepareSurfaceLocked()函数中，是通过Surface.setLayer()完成的。在4.5节会深入为大家揭开WMS动画子系统的面纱。

4.3.4 关于显示次序的小结

这一节讨论了窗口类型对窗口显示次序的影响。窗口根据自己的类型得出其主序及子序，然后addWindowToListInOrderLocked()根据主序、子序以及其所属的Activity的顺序，按照升序排列在DisplayContent的mWindows列表中。然后assignLayersLocked()为mWindows中的所有窗口分配最终的显示次序。之后，WMS的动画系统将最终的显示次序通过Surface.setLayer()设置进SurfaceFlinger。

> [1] 关于Choreographer，请参考邓凡平的博客《Android Project Butter分析》（http://blog.csdn.net/innost/article/details/8272867）。
>
> [2] 读者可阅读《深入理解Android 卷I》第4章“深入理解Zygote”来了解和zygote相关的知识
>
> [3] 关于Wi-Fi Display的详细信息，请读者参考http://blog.csdn.net/innost/article/details/8474683的介绍。

第5章 深入理解Android输入系统（节选）

本章主要内容：

- 研究输入事件从设备节点开始到窗口处理函数的流程
- 介绍原始输入事件的读取与加工的原理
- 研究事件派发机制
- 讨论事件在输入系统与窗口之间的传递与反馈的过程
- 介绍焦点窗口的选择、ANR的产生以及以软件方式模拟用户操作的原理

本章涉及的源代码文件名及位置：

- SystemServer.java

frameworks\base\services\java\com\android\server\SystemServer.java

- InputManagerService.java

frameworks\base\services\java\com\android\server\input/InputManagerService.java

- WindowManagerService.java

frameworks\base\services\java\com\android\server\wm\WindowManagerService.java

- WindowState.java

frameworks\base\services\java\com\android\server\wm\WindowState.java

- InputMonitor.java

frameworks\base\services\java\com\android\server\wm\InputMonitor.java

- InputEventReceiver.java

frameworks\base\core\java\android\view\InputEventReceiver.java

- com_android_server_input_InputManagerService.cpp

frameworks\base\services\jni\com_android_server_input_InputManagerService.cpp

- android_view_InputEventReceiver.cpp

frameworks\base\core\jni\android_view_InputEventReceiver.cpp

- InputManager.cpp

frameworks\base\services\input\InputManager.cpp

- EventHub.cpp

frameworks\base\services\input\EventHub.cpp

- EventHub.h

frameworks\base\services\input\EventHub.h

- InputDispatcher.cpp

frameworks\base\services\input\InputDispatcher.cpp

- InputDispatcher.h

frameworks\base\services\input\InputDispatcher.h

- InputTransport.cpp

frameworks\base\libs\androidfw\InputTransport.cpp

- InputTransport.h

frameworks\base\include\androidfw\InputTransport.h

5.1 初识Android输入系统

第4章通过分析WMS详细讨论了Android的窗口管理、布局及动画的工作机制。窗口不仅是内容绘制的载体，同时也是用户输入事件的目标。本章将详细讨论Android输入系统的工作原理，包括输入设备的管理、输入事件的加工方式以及派发流程。因此本章的探讨对象有两个：输入设备、输入事件。

触摸屏与键盘是Android最普遍也是最标准的输入设备。其实Android所支持的输入设备的种类不止这两个，鼠标、游戏手柄均在内建的支持之列。当输入设备可用时，Linux内核会在/dev/input/下创建对应的名为event0~n或其他名称的设备节点。而当输入设备不可用时，则会将对应的节点删除。在用户空间可以通过ioctl的方式从这些设备节点中获取其对应的输入设备的类型、厂商、描述等信息。

当用户操作输入设备时，Linux内核接收到相应的硬件中断，然后将中断加工成原始的输入事件数据并写入其对应的设备节点中，在用户空间可以通过read()函数将事件数据读出。

Android输入系统的工作原理概括来说，就是监控/dev/input/下的所有设备节点，当某个节点有数据可读时，将数据读出并进行一系列的翻译加工，然后在所有的窗口中寻找合适的事件接收者，并派发给它。

以Nexus4为例，其/dev/input/下有evnet0~5六个输入设备的节点。它们都是什么输入设备呢？用户的一次输入操作会产生什么样的事件数据呢？获取答案的最简单的办法就是是用getevent与sendevent工具。

5.1.1 getevent与sendevent工具

Android系统提供了getevent与sendevent两个工具供开发者从设备节点中直接读取输入事件或写入输入事件。

getevent监听输入设备节点的内容，当输入事件被写入到节点中时，getevent会将其读出并打印在屏幕上。由于getevent不会对事件数据做任何加工，因此其输出的内容是由内核提供的最原始的事件。其用法如下：

```
adb shell getevent [-选项] [device_path]
```

其中device_path是可选参数，用以指明需要监听的设备节点路径。如果省略此参数，则监听所有设备节点的事件。

打开模拟器，执行adb shell getevent –t（-t参数表示打印事件的时间戳），并按一下电源键（不要松手），可以得到以下一条输出，输出的部分数值会因机型的不同而有所差异，但格式一致：

```
[   1262.443489] /dev/input/event0: 0001 0074 00000001
```

松开电源键时，又会产生以下一条输出：

```
[   1262.557130] /dev/input/event0: 0001 0074 00000000
```

这两条输出便是按下和抬起电源键时由内核生成的原始事件。注意其输出是十六进制的。每条数据有五项信息：产生事件时的时间戳（[1262.443489]），产生事件的设备节点（/dev/input/event0），事件类型（0001），事件代码（0074）以及事件的值（00000001）。其中时间戳、类型、代码、值便是原始事件的四项基本元素。除时间戳外，其他三项元素的实际意义依照设备类型及厂商的不同而有所区别。在本例中，类型0x01表示此事件为一条按键事件，代码0x74表示电源键的扫描码，值0x01表示按下，0x00则表示抬起。这两条原始数据被输入系统包装成两个KeyEvent对象，作为两个按键事件派发给Framework中感兴趣的模块或应用程序。

注意 一条原始事件所包含的信息量是比较有限的。而在Android API中所使用的某些输入事件，如触摸屏点击/滑动，包含了很多的信息，如XY坐标，触摸点索引等，其实是输入系统整合了多个原始事件后的结果。这个过程将在5.2.4节中详细探讨。

为了对原始事件有一个感性的认识，读者可以在运行getevent的过程中尝试一下其他的输入操作，观察一下每种输入所对应的设备节点及四项元素的取值。

输入设备的节点不仅在用户空间可读，而且是可写的，因此可以将将原始事件写入到节点中，从而实现模拟用户输入的功能。sendevent工具的作用正是如此。其用法如下：

```
sendevent <节点路径> <类型><代码> <值>
```

可以看出，sendevent的输入参数与getevent的输出是对应的，只不过sendevent的参数为十进制。电源键的代码0x74的十进制为116，因此可以通过快速执行如下两条命令实现点击电源键的效果：

```
adb shell sendevent /dev/input/event0 1 116 1 #按下电源键
adb shell sendevent /dev/input/event0 1 116 0 #抬起电源键
```

执行完这两条命令后，可以看到设备进入了休眠或被唤醒，与按下实际的电源键的效果一模一样。另外，执行这两条命令的时间间隔便是用户按住电源键所保持的时间，所以如果执行第一条命令后迟迟不执行第二条，则会产生长按电源键的效果——关机对话框出现了。很有趣不是么？输入设备节点在用户空间可读可写的特性为自动化测试提供了一条高效的途径。[1]

现在，读者对输入设备节点以及原始事件有了直观的认识，接下来看一下Android输入系统的基本原理。

5.1.2 Android输入系统简介

上一节讲述了输入事件的源头是位于/dev/input/下的设备节点，而输入系统的终点是由WMS管理的某个窗口。最初的输入事件为内核生成的原始事件，而最终交付给窗口的则是KeyEvent或MotionEvent对象。因此Android输入系统的主要工作是读取设备节点中的原始事件，将其加工封装，然后派发给一个特定的窗口以及窗口中的控件。这个过程由InputManagerService（以下简称IMS）系统服务为核心的多个参与者共同完成。

输入系统的总体流程和参与者如图5-1所示。

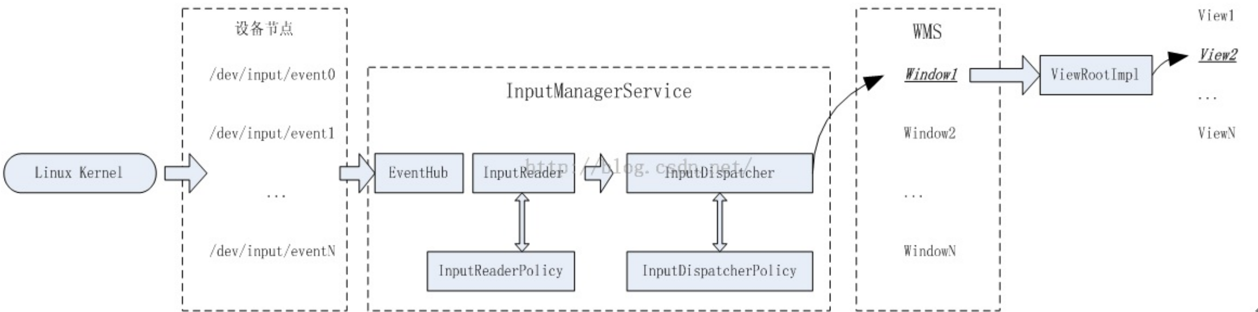


图 5-1 输入系统的总体流程与参与者

图5-1描述了输入事件的处理流程以及输入系统中最基本的参与者。它们是：

- Linux内核，接受输入设备的中断，并将原始事件的数据写入到设备节点中。
- 设备节点，作为内核与IMS的桥梁，它将原始事件的数据暴露给用户空间，以便IMS可以从中读取事件。
- InputManagerService，一个Android系统服务，它分为Java层和Native层两部分。Java层负责与WMS的通信。而Native层则是InputReader和InputDispatcher两个输入系统关键组件的运行容器。
- EventHub，直接访问所有的设备节点。并且正如其名字所描述的，它通过一个名为getEvents()的函数将所有输入系统相关的待处理的底层事件返回给使用者。这些事件包括原始输入事件、设备节点的增删等。
- InputReader，I是IMS中的关键组件之一。它运行于一个独立的线程中，负责管理输入设备的列表与配置，以及进行输入事件的加工处理。它通过其线程循环不断地通过getEvents()函数从EventHub中将事件取出并进行处理。对于设备节点的增删事件，它会更新输入设备列表于配置。对于原始输入事件，InputReader对其进行翻译、组装、封装为包含了更多信息、更具可读性的输入事件，然后交给InputDispatcher进行派发。
- InputReaderPolicy，它为InputReader的事件加工处理提供一些策略配置，例如键盘布局信息等。
- InputDispatcher，是IMS中的另一个关键组件。它也运行于一个独立的线程中。InputDispatcher中保管了来自WMS的所有窗口的信息，其收到来自InputReader的输入事件后，会在其保管的窗口中寻找合适的窗口，并将事件派发给此窗口。
- InputDispatcherPolicy，它为InputDispatcher的派发过程提供策略控制。例如截取某些特定的输入事件用作特殊用途，或者阻止将某些事件派发给目标窗口。一个典型的例子就是HOME键被InputDispatcherPolicy截取到PhoneWindowManager中进行处理，并阻止窗口收到HOME键按下的事件。
- WMS，虽说不是输入系统中的一员，但是它却对InputDispatcher的正常工作起到了至关重要的作用。当新建窗口时，WMS为新窗口和IMS创建了事件传递所用的通道。另外，WMS还将所有窗口的信息，包括窗口的可点击区域，焦点窗口等信息，实时地更新到IMS的InputDispatcher中，使得InputDispatcher可以正确地将事件派发到指定的窗口。
- ViewRootImpl，对于某些窗口，如壁纸窗口、SurfaceView的窗口来说，窗口即是输入事件派发的终点。而对于其他的如Activity、对话框等使用了Android控件系统的窗口来说，输入事件的终点是控件（View）。ViewRootImpl将窗口所接收到的输入事件沿着控件树将事件派发给感兴趣的控件。

简单来说，内核将原始事件写入到设备节点中，InputReader不断地通过EventHub将原始事件取出来并翻译加工成Android输入事件，然后交给InputDispatcher。InputDispatcher根据WMS提供的窗口信息将事件交给合适的窗口。窗口的ViewRootImpl对象再沿着控件树将事件派发给感兴趣的控件。控件对其收到的事件作出响应，更新自己的画面、执行特定的动作。所有这些参与者以IMS为核心，构建了Android庞大而复杂的输入体系。

Linux内核对硬件中断的处理超出了本书的讨论范围，因此本章将以IMS为重点，详细讨论除Linux内核以外的其他参与者的工作原理。

5.1.3 IMS的构成

同以往一样，本节通过IMS的启动过程，探讨IMS的构成。上一节提到，IMS分为Java层与Native层两个部分，其启动过程是从Java部分的初始化开始，进而完成Native部分的初始化。

1. IMS的诞生

同其他系统服务一样，IMS在SystemServer中的ServerThread线程中启动。

```
[SystemServer.java-->ServerThread.run()]
public void run() {
     ......
    InputManagerService inputManager = null;
     ......
     // **① 新建IMS对象。**注意第二个参数wmHandler，这说明IMS的一部分功能可能会在WMS的线程中完成
     inputManager= new InputManagerService(context, wmHandler);
     // 将IMS发布给ServiceManager，以便其他人可以访问IMS提供的接口
     ServiceManager.addService(Context.INPUT_SERVICE,inputManager);
     // 设置向WMS发起回调的callback对象
     inputManager.setWindowManagerCallbacks(wm.getInputMonitor());
     // **② 正式启动IMS**
    inputManager.start();
     ......
     /* 设置IMS给DisplayManagerService。DisplayManagerService将会把屏幕的信息发送给输入
        系统作为事件加工的依据。在5.2.4节将会讨论到这些信息的作用 */
    display.setInputManager(inputManager);
}
```

IMS的诞生分为两个阶段：

- 创建新的IMS对象。
- 调用IMS对象的start()函数完成启动。

（1） IMS的创建

IMS的构造函数如下：

```
[InputManagerService.java-->InputManagerService.InputManagerService()]
public InputManagerService(Context context,Handler handler) {
    /* 使用wmHandler的Looper新建一个InputManagerHandler。InputManagerHandler将运行在
       WMS的主线程中*/
   this.mHandler = new InputManagerHandler(handler.getLooper());
    ......
    // 每一个分为Java和Native两部分的对象在创建时都会有一个nativeInput函数
    mPtr =nativeInit(this, mContext, mHandler.getLooper().getQueue());
}
```

可以看出，IMS的构造函数非常简单。看来绝大部分的初始化工作都位于Native层。参考nativeInit()函数的实现。

```
[com_android_server_input_InputManagerService.cpp-->nativeInit()]
static jint nativeInit(JNIEnv* env, jclass clazz,
       jobject serviceObj, jobject contextObj, jobject messageQueueObj) {
   sp&lt;MessageQueue&gt; messageQueue =android_os_MessageQueue_getMessageQueue(env, mess
    /* 新建了一个NativeInputManager对象，NativeInputManager，此对象将是Native层组件与
      Java层IMS进行通信的桥梁 */
   NativeInputManager* im = new NativeInputManager(contextObj, serviceObj,
           messageQueue->getLooper());
   im->incStrong(serviceObj);
    // 返回了NativeInputManager对象的指针给Java层的IMS，IMS将其保存在mPtr成员变量中
    returnreinterpret_cast&lt;jint&gt;(im);
}
```

nativeInit()函数创建了一个类型为NativeInputManager的对象，它是Java层与Native层互相通信的桥梁。

看下这个类的声明可以发现，它实现了InputReaderPolicyInterface与InputDispatcherPolicyInterface两个接口。这说明上一节曾经介绍过的两个重要的输入系统参与者InputReaderPolicy和InputDispatcherPolicy是由NativeInputManager实现的，然而它仅仅为两个策略提供接口实现而已，并不是策略的实际实现者。NativeInputManager通过JNI回调Java层的IMS，由它完成决策。这一小节暂不讨论其实现细节，读者只要先记住两个策略参与者的接口实现位于NativeInputManager即可。

接下来看一下NativeInputManager的创建：

```
[com_android_server_input_InputManagerService.cpp
-->NativeInputManager::NativeInputManager()]
NativeInputManager::NativeInputManager(jobjectcontextObj,
       jobject serviceObj, const sp&lt;Looper&gt;& looper) :
       mLooper(looper) {
    ......
    // 出现重点了， NativeInputManager创建了EventHub
   sp&lt;EventHub&gt; eventHub = new EventHub();
    // 接着创建了Native层的InputManager
   mInputManager = new InputManager(eventHub, this, this);
}
```

在NativeInputManager的构造函数中，创建了两个关键人物，分别是EventHub与InputManager。EventHub复杂的构造函数使其在创建后便拥有了监听设备节点的能力，这一小节中暂不讨论它的构造函数，读者仅需知道EventHub在这里初始化即可。紧接着便是

InputManager的创建了，看一下其构造函数：

```
[InputManager.cpp-->InputManager::InputManager()]
InputManager::InputManager(
        const sp&lt;EventHubInterface&gt;& eventHub,
        const sp&lt;InputReaderPolicyInterface&gt;& readerPolicy,
        const sp&lt;InputDispatcherPolicyInterface&gt;& dispatcherPolicy) {
    // 创建InputDispatcher
   mDispatcher = new InputDispatcher(dispatcherPolicy);
    // 创建 InputReader
    mReader= new InputReader(eventHub, readerPolicy, mDispatcher);
    // 初始化
   initialize();
}
```

再看initialize()函数：

```
[InputManager.cpp-->InputManager::initialize()]
void InputManager::initialize() {
    // 创建供InputReader运行的线程InputReaderThread
   mReaderThread = new InputReaderThread(mReader);
    // 创建供InputDispatcher运行的线程InputDispatcherThread
   mDispatcherThread = new InputDispatcherThread(mDispatcher);
}
```

InputManager的构造函数也比较简洁，它创建了四个对象，分别为IMS的核心参与者InputReader与InputDispatcher，以及它们所在的线程InputReaderThread与InputDispatcherThread。注意InputManager的构造函数的参数readerPolicy与dispatcherPolicy，它们都是NativeInputManager。

至此，IMS的创建完成了。在这个过程中，输入系统的重要参与者均完成创建，并得到了如图5-2所描述的一套体系。

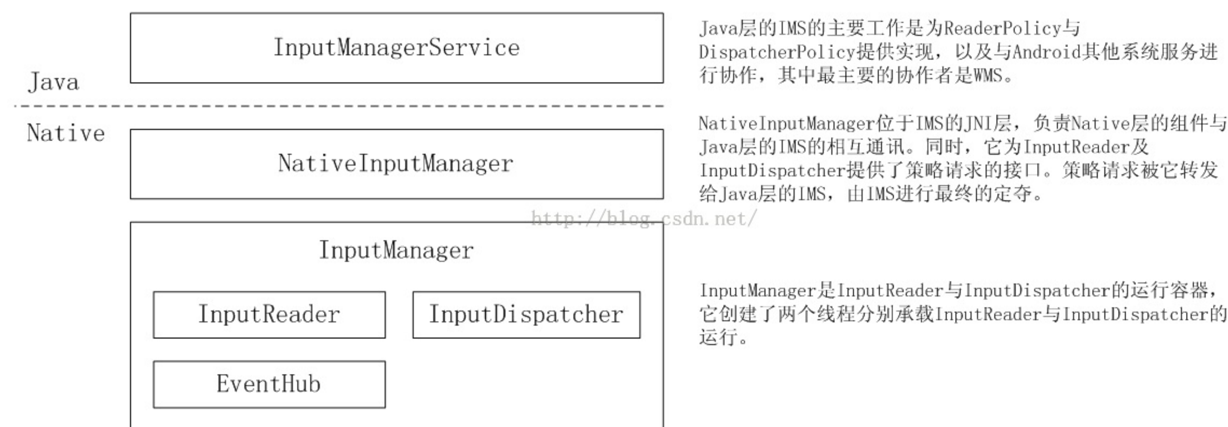


图 5-2 IMS的结构体系

（2） IMS的启动与运行

完成IMS的创建之后，ServerThread执行了InputManagerService.start()函数以启动IMS。InputManager的创建过程分别为InputReader与InputDispatcher创建了承载它们运行的线程，然而并未将这两个线程启动，因此IMS的各员大将仍处于待命状态。此时start()函数的功能就是启动这两个线程，使得InputReader于InputDispatcher开始工作。

当两个线程启动后，InputReader在其线程循环中不断地从EventHub中抽取原始输入事件，进行加工处理后将加工所得的事件放入InputDispatcher的派发发队列中。InputDispatcher则在其线程循环中将派发队列中的事件取出，查找合适的窗口，将事件写入到窗口的事件接收管道中。窗口事件接收线程的Looper从管道中将事件取出，交由事件处理函数进行事件响应。整个过程共有三个线程首尾相接，像三台水泵似的一层层地将事件交付给事件处理函数。如图5-3所示。

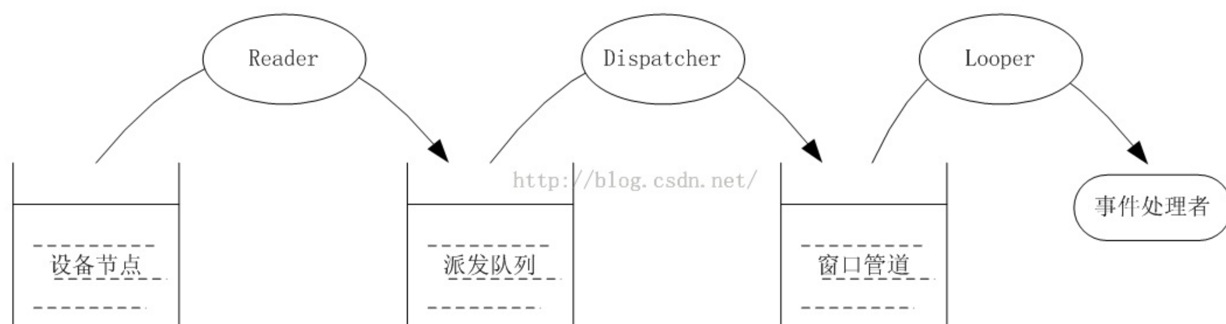


图 5-3 三个线程，三台水泵

InputManagerService.start()函数的作用，就像为Reader线程、Dispatcher线程这两台水泵按下开关，而Looper这台水泵在窗口创建时便已经处于运行状态了。自此，输入系统动力十足地开始运转，设备节点中的输入事件将被源源不断地抽取给事件处理者。本章的主要内容便是讨论这三台水泵的工作原理。

2. IMS的成员关系

根据对IMS的创建过程的分析，可以得到IMS的成员关系如图5-4所示，这幅图省略了一些非关键的引用与继承关系。

注意 IMS内部做了很多的抽象工作，EventHub、nputReader以及InputDispatcher等实际上都继承自相应的名为XXXInterface的接口，并且仅通过接口进行相互之间的引用。鉴于这些接口各自仅有唯一的实现，为了简化叙述我们将不提及这些接口，但是读者在实际学习与研究时需要注意这一点。

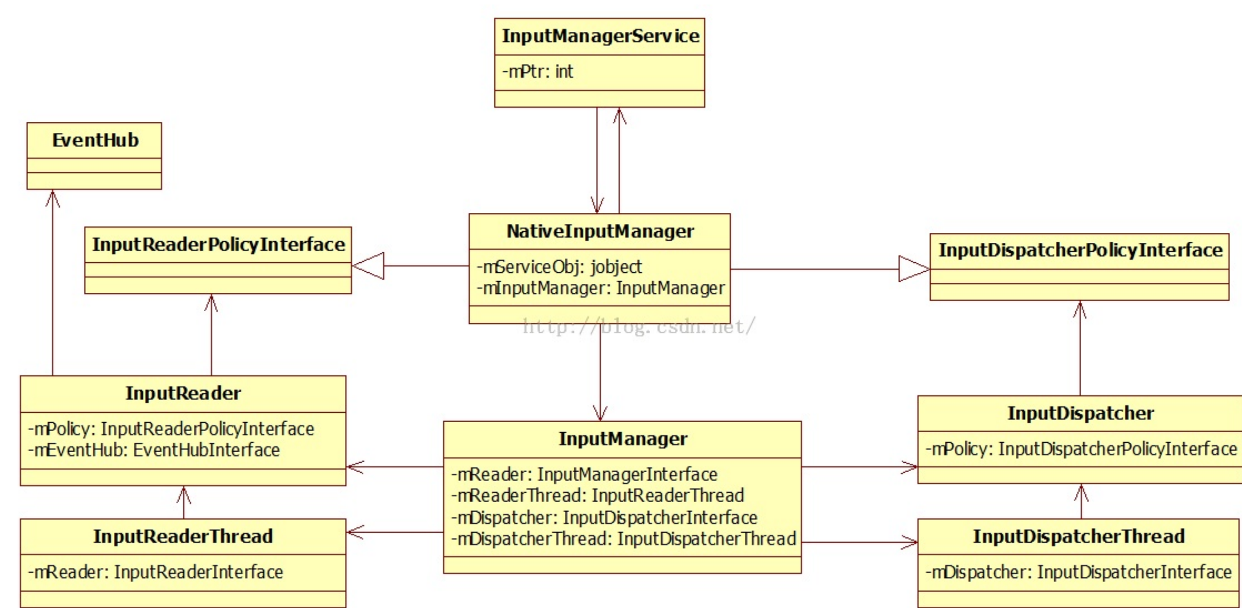


图 5-4 IMS的成员关系

在图5-4中，左侧部分为Reader子系统对应于图5-3中的第一台水泵，右侧部分为Dispatcher子系统，对应于图5-3中的第二台水泵。了解了IMS的成员关系后便可以开始我们的IMS深入理解之旅了！

5.2 原始事件的读取与加工

本节将深入探讨第一台水泵——Reader子系统的工作原理。Reader子系统的输入端是设备节点，输出端是Dispatcher子系统的派发队列。从设备节点到派发队列之间的过程发生了什么呢？本章一开始曾经介绍过，一个设备节点对应了一个输入设备，并且其中存储了内核写入的原始事件。因此设备节点拥有两个概念：设备与原始事件。因此Reader子系统需要处理输入设备以及原始事件两种类型的对象。

设备节点的新建与删除表示了输入设备的可用与无效，Reader子系统需要加载或删除对应设备的配置信息；而设备节点中是否有内容可读表示了是否有新的原始事件到来，有新的原始事件到来时Reader子系统需要开始对新事件进行加工并放置到派发队列中。问题是应该如何监控设备节点的新建与删除动作以及如何确定节点中有内容可读呢？最简单的办法是在线程循环中不断地轮询，然而这会导致非常低下的效率，更会导致电量在无谓地轮询中消耗。Android使用由Linux提供的两套机制INotify与Epoll优雅地解决了这两个问题。在正式探讨Reader子系统的工作原理之前，需要首先了解这两套机制的使用方法。

5.2.1 基础知识：INotify与Epoll

1. INotify介绍与使用

INotify是一个Linux内核所提供的一种文件系统变化通知机制。它可以为应用程序监控文件系统的变化，如文件的新建、删除、读写等。INotify机制有两个基本对象，分别为inotify对象与watch对象，都使用文件描述符表示。

inotify对象对应了一个队列，应用程序可以向inotify对象添加多个监听。当被监听的事件发生时，可以通过read()函数从inotify对象中将事件信息读取出来。Inotify对象可以通过以下方式创建：

```
int inotifyFd = inotify_init();
```

而watch对象则用来描述文件系统的变化事件的监听。它是一个二元组，包括监听目标和事件掩码两个元素。监听目标是文件系统的一个路径，可以是文件也可以是文件夹。而事件掩码则表示了需要需要监听的事件类型，掩码中的每一位代表一种事件。可以监听的事件种类很多，其中就包括文件的创建(IN_CREATE)与删除(IN_DELETE)。读者可以参阅相关资料以了解其他可监听的事件种类。以下代码即可将一个用于监听输入设备节点的创建与删除的watch对象添加到inotify对象中：

```
int wd = inotify_add_watch (inotifyFd, "/dev/input",IN_CREATE | IN_DELETE);
```

完成上述watch对象的添加后，当/dev/input/下的设备节点发生创建与删除操作时，都会将相应的事件信息写入到inotifyFd所描述的inotify对象中，此时可以通过read()函数从inotifyFd描述符中将事件信息读取出来。

事件信息使用结构体inotify_event进行描述：

```
struct inotify_event {
       __s32          wd;            /* 事件对应的Watch对象的描述符 */
       __u32          mask;          /* 事件类型，例如文件被删除，此处值为IN_DELETE */
       __u32          cookie;
       __u32          len;           /* name字段的长度 */
       char           name[0];       /* 可变长的字段，用于存储产生此事件的文件路径*/
};
```

当没有监听事件发生时，可以通过如下方式将一个或多个未读取的事件信息读取出来：

```
size_t len = read (inotifyFd, events_buf,BUF_LEN);
```

其中events_buf是inotify_event的数组指针，能够读取的事件数量由取决于数组的长度。成功读取事件信息后，便可根据inotify_event结构体的字段判断事件类型以及产生事件的文件路径了。

总结一下INotify机制的使用过程：

- 通过inotify_init()创建一个inotify对象。

- 通过inotify_add_watch将一个或多个监听添加到inotify对象中。
- 通过read()函数从inotify对象中读取监听事件。当没有新事件发生时，inotify对象中无任何可读数据。

通过INotify机制避免了轮询文件系统的麻烦，但是还有一个问题，INotify机制并不是通过回调的方式通知事件，而需要使用者主动从inotify对象中进行事件读取。那么何时才是读取的最佳时机呢？这就需要借助Linux的另一个优秀的机制Epoll了。

2. Epoll介绍与使用

无论是从设备节点中获取原始输入事件还是从inotify对象中读取文件系统事件，都面临一个问题，就是这些事件都是偶发的。也就是说，大部分情况下设备节点、inotify对象这些文件描述符中都是无数据可读的，同时又希望有事件到来时可以尽快地对事件作出反应。为解决这个问题，我们不希望不断地轮询这些描述符，也不希望为每个描述符创建一个单独的线程进行阻塞时的读取，因为这都将会导致资源的极大浪费。

此时最佳的办法是使用Epoll机制。Epoll可以使用一次等待监听多个描述符的可读/可写状态。等待返回时携带了可读的描述符或自定义的数据，使用者可以据此读取所需的数据后可以再次进入等待。因此不需要为每个描述符创建独立的线程进行阻塞读取，避免了资源浪费的同时又可以获得较快的响应速度。

Epoll机制的接口只有三个函数，十分简单。

- epoll_create(int max_fds)：创建一个epoll对象的描述符，之后对epoll的操作均使用这个描述符完成。max_fds参数表示了此epoll对象可以监听的描述符的最大数量。
- epoll_ctl (int epfd, int op,int fd, struct epoll_event *event)：用于管理注册事件的函数。这个函数可以增加/删除/修改事件的注册。
- int epoll_wait(int epfd, structepoll_event * events, int maxevents, int timeout)：用于等待事件的到来。当此函数返回时，events数组参数中将会包含产生事件的文件描述符。

接下来以监控若干描述符可读事件为例介绍一下epoll的用法。

（1） 创建epoll对象

首先通过epoll_create()函数创建一个epoll对象：

```
Int epfd = epoll_create(MAX_FDS)
```

（2） 填充epoll_event结构体

接着为每一个需监控的描述符填充epoll_event结构体，以描述监控事件，并通过epoll_ctl()函数将此描述符与epoll_event结构体注册进epoll对象。epoll_event结构体的定义如下:

```
struct epoll_event {
    __uint32_tevents; /* 事件掩码，指明了需要监听的事件种类*/
   epoll_data_t data; /* 使用者自定义的数据，当此事件发生时该数据将原封不动地返回给使用者 */
};
```

epoll_data_t联合体的定义如下，当然，同一时间使用者只能使用一个字段：

```
typedef union epoll_data {
    void*ptr;
    int fd;
   __uint32_t u32;
   __uint64_t u64;
} epoll_data_t;
```

epoll_event结构中的events字段是一个事件掩码，用以指明需要监听的事件种类，同INotify一样，掩码的每一位代表了一种事件。常用的事件有EPOLLIN（可读），EPOLLOUT（可写），EPOLLERR（描述符发生错误），EPOLLHUP（描述符被挂起）等。更多支持的事件读者可参考相关资料。

data字段是一个联合体，它让使用者可以将一些自定义数据加入到事件通知中，当此事件发生时，用户设置的data字段将会返回给使用者。在实际使用中常设置epoll_event.data.fd为需要监听的文件描述符，事件发生时便可以根据epoll_event.data.fd得知引发事件的描述符。当然也可以设置epoll_event.data.fd为其他便于识别的数据。

填充epoll_event的方法如下：

```
    structepoll_event eventItem;
   memset(&eventItem, 0, sizeof(eventItem));
   eventItem.events = EPOLLIN | EPOLLERR | EPOLLHUP; // 监听描述符可读以及出错的事件
    eventItem.data.fd= listeningFd; // 填写自定义数据为需要监听的描述符
```

接下来就可以使用epoll_ctl()将事件注册进epoll对象了。epoll_ctl()的参数有四个：

- epfd是由epoll_create()函数所创建的epoll对象的描述符。
- op表示了何种操作，包括EPOLL_CTL_ADD/DEL/MOD三种，分别表示增加/删除/修改注册事件。
- fd表示了需要监听的描述符。
- event参数是描述了监听事件的详细信息的epoll_event结构体。

注册方法如下：

```
    // 将事件监听添加到epoll对象中去
    result =epoll_ctl(epfd, EPOLL_CTL_ADD, listeningFd, &eventItem);
```

重复这个步骤可以将多个文件描述符的多种事件监听注册到epoll对象中。完成了监听的注册之后，便可以通过epoll_wait()函数等待事件的到来了。

（3） 使用epoll_wait()函数等待事件

epoll_wait()函数将会使调用者陷入等待状态，直到其注册的事件之一发生之后才会返回，并且携带了刚刚发生的事件的详细信息。其签名如下：

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

- epfd是由epoll_create()函数所创建的epoll对象描述符。
- events是一个epoll_event的数组，此函数返回时，事件的信息将被填充至此。
- maxevents表示此次调用最多可以获取多少个事件，当然，events参数必须能够足够容纳这么多事件。
- timeout表示等待超时的事件。

epoll_wait()函数返回值表示获取了多少个事件。

4. 处理事件

epoll_wait返回后，便可以根据events数组中所保存的所有epoll_event结构体的events字段与data字段识别事件的类型与来源。

Epoll的使用步骤总结如下：

- 通过epoll_create()创建一个epoll对象。
- 为需要监听的描述符填充epoll_events结构体，并使用epoll_ctl()注册到epoll对象中。
- 使用epoll_wait()等待事件的发生。
- 根据epoll_wait()返回的epoll_events结构体数组判断事件的类型与来源并进行处理。
- 继续使用epoll_wait()等待新事件的发生。

3. INotify与Epoll的小结

INotify与Epoll这两套由Linux提供的事件监听机制以最小的开销解决了文件系统变化以及文件描述符可读可写状态变化的监听问题。它们是Reader子系统运行的基石，了解了这两个机制的使用方法之后便为对Reader子系统的分析学习铺平了道路。

5.2.2 InputReader的总体流程

在了解了INotify与Epoll的基础知识之后便可以正是开始分析Reader子系统的工作原理了。首先要理解InputReader的运行方式。在5.1.3节介绍了InputReader被InputManager创建，并运行于InputReaderThread线程中。InputReader如何在InputReaderThread中运行呢？

InputReaderThread继承自C++的Thread类，Thread类封装了pthread线程工具，提供了与Java层Thread类相似的API。C++的Thread类提供了一个名为threadLoop()的纯虚函数，当线程开始运行后，将会在内建的线程循环中不断地调用threadLoop()，直到此函数返回false，则退出线程循环，从而结束线程。

InputReaderThread仅仅重写了threadLoop()函数：

```
[InputReader.cpp-->InputReaderThread::threadLoop()]
bool InputReaderThread::threadLoop() {
   mReader->loopOnce(); // 执行InputReader的loopOnce()函数
    returntrue;
}
```

InputReaderThread启动后，其线程循环将不断地执行InputReader.loopOnce()函数。因此这个loopOnce()函数作为线程循环的循环体包含了InputReader的所有工作。

注意 C++层的Thread类与Java层的Thread类有着一个显著的不同。C++层Thread类内建了线程循环，threadLoop()就是一次循环而已，只要返回值为true，threadLoop()将会不断地被内建的循环调用。这也是InputReader.loopOnce()函数名称的由来。而Java层Thread类的run()函数则是整个线程的全部，一旦其退出，线程也便完结。

接下来看一下InputReader.loopOnce()的代码，分析一下InputReader在一次线程循环中做了什么。

```
[InputReader.cpp-->InputReader::loopOnce()]
void InputReader::loopOnce() {
    ......
    /* **① 通过EventHub抽取事件列表**。读取的结果存储在参数mEventBuffer中，返回值表示事件的个数
       当EventHub中无事件可以抽取时，此函数的调用将会阻塞直到事件到来或者超时 */
    size_tcount = mEventHub->getEvents(timeoutMillis
                                           ,mEventBuffer, EVENT_BUFFER_SIZE);
    {
       AutoMutex _l(mLock);
        ......
        if(count) {
           // **② 如果有抽得事件，则调用processEventsLocked()函数对事件进行加工处理**
           processEventsLocked(mEventBuffer, count);
        }
        ......
    }
    ......
    /* **③ 发布事件。** processEventsLocked()函数在对事件进行加工处理之后，便将处理后的事件存储在
      mQueuedListener中。在循环的最后，通过调用flush()函数将所有事件交付给InputDispatcher */
   mQueuedListener->flush();
}
```

InputReader的一次线程循环的工作思路非常清晰，一共三步：

- 首先从EventHub中抽取未处理的事件列表。这些事件分为两类，一类是从设备节点中读取的原始输入事件，另一类则是输入设备可用性变化事件，简称为设备事件。

- 通过processEventsLocked()对事件进行处理。对于设备事件，此函数对根据设备的可用性加载或移除设备对应的配置信息。对于原始输入事件，则在进行转译、封装与加工后将结果暂存到mQueuedListener中。
- 所有事件处理完毕后，调用mQueuedListener.flush()将所有暂存的输入事件一次性地交付给InputDispatcher。

这便是InputReader的总体工作流程。而我们接下来将详细讨论这三步的实现。

5.2.3 深入理解EventHub

InputReader在其线程循环中的第一个工作便是从EventHub中读取一批未处理的事件。EventHub是如何工作的呢？

EventHub的直译是事件集线器，顾名思义，它将所有的输入事件通过一个接口getEvents()将从多个输入设备节点中读取的事件交给InputReader，是输入系统最底层的一个组件。它是如何工作呢？没错，正是基于前文所述的INotify与Epoll两套机制。

1. 设备节点监听的建立

在EventHub的构造函数中，它通过INotify与Epoll机制建立起了对设备节点增删事件以及可读状态的监听。在继续之前，请读者先回忆一下INotify与Epoll的使用方法。

EventHub的构造函数如下：

```
[EventHub.cpp-->EventHub::EventHub()]
EventHub::EventHub(void) :
       mBuiltInKeyboardId(NO_BUILT_IN_KEYBOARD), mNextDeviceId(1),
       mOpeningDevices(0), mClosingDevices(0),
       mNeedToSendFinishedDeviceScan(false),
       mNeedToReopenDevices(false), mNeedToScanDevices(true),
       mPendingEventCount(0), mPendingEventIndex(0), mPendingINotify(false) {
    /* **① 首先使用epoll_create()函数创建一个epoll对象**。EPOLL_SIZE_HINT指定最大监听个数为8
       这个epoll对象将用来监听设备节点是否有数据可读（有无事件） */
    mEpollFd= epoll_create(EPOLL_SIZE_HINT);
    // **② 创建一个inotify对象**。这个inotify对象将被用来监听设备节点的增删事件
   mINotifyFd = inotify_init();
    /* 将存储设备节点的路径/dev/input作为监听对象添加到inotify对象中。当此文件夹下的设备节点
       发生创建与删除事件时，都可以通过mINotifyFd读取事件的详细信息 */
    intresult = inotify_add_watch(mINotifyFd, DEVICE_PATH, IN_DELETE | IN_CREATE);
    /* **③ 接下来将mINotifyFd作为epoll的一个监控对象**。当inotify事件到来时，epoll_wait()将
       立刻返回，EventHub便可从mINotifyFd中读取设备节点的增删信息，并作相应处理 */
    structepoll_event eventItem;
   memset(&eventItem, 0, sizeof(eventItem));
   eventItem.events = EPOLLIN; // 监听mINotifyFd可读
    // 注意这里并没有使用fd字段，而使用了自定义的值EPOLL_ID_INOTIFY
   eventItem.data.u32 = EPOLL_ID_INOTIFY;
    // 将对mINotifyFd的监听注册到epoll对象中
    result =epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mINotifyFd, &eventItem);
    /* 在构造函数剩余的代码中，EventHub创建了一个名为wakeFds的匿名管道，并将管道读取端的描述符
       的可读事件注册到epoll对象中。因为InputReader在执行getEvents()时会因无事件而导致其线程
       阻塞在epoll_wait()的调用里，然而有时希望能够立刻唤醒InputReader线程使其处理一些请求。此
       时只需向wakeFds管道的写入端写入任意数据，此时读取端有数据可读，使得epoll_wait()得以返回，
       从而达到唤醒InputReader线程的目的*/
    ......
}
```

EventHub的构造函数初识化了Epoll对象和INotify对象，分别监听原始输入事件与设备节点增删事件。同时将INotify对象的可读性事件也注册到Epoll中，因此EventHub可以像处理原始输入事件一样监听设备节点增删事件了。

构造函数同时也揭示了EventHub的监听工作分为设备节点和原始输入事件两个方面，接下来将深入探讨这两方面的内容。

2. getEvents()函数的工作方式

正如前文所述，InputReaderThread的线程循环为Reader子系统提供了运转的动力，EventHub的工作也是由它驱动的。InputReader::loopOnce()函数调用EventHub::getEvents()函数获取事件列表，所以这个getEvents()是EventHub运行的动力所在，几乎包含了EventHub的所有工作内容，因此首先要将getEvents()函数的工作方式搞清楚。

getEvents()函数的签名如下：

size_t EventHub::getEvents(int timeoutMillis,RawEvent* buffer, size_t bufferSize)

此函数将尽可能多地读取设备增删事件与原始输入事件，将它们封装为RawEvent结构体，并放入buffer中供InputReader进行处理。RawEvent结构体的定义如下：

```
[EventHub.cpp-->RawEvent]
struct RawEvent {
    nsecs_twhen;              /* 发生事件时的时间戳 */
    int32_tdeviceId;         /* 产生事件的设备Id，它是由EventHub自行分配的，InputReader
                                   以根据它从EventHub中获取此设备的详细信息 */
    int32_ttype;              /* 事件的类型 */
    int32_tcode;              /* 事件代码 */
    int32_tvalue;             /* 事件值 */
};
```

可以看出，RawEvent结构体与getevent工具的输出十分一致，包含了原始输入事件的四个基本元素，因此用RawEvent结构体表示原始输入事件是非常直观的。RawEvent同时也用来表示设备增删事件，为此，EventHub定义了三个特殊的事件类型DEVICE_ADD、DEVICE_REMOVED以及FINISHED_DEVICE_SCAN，用以与原始输入事件进行区别。

由于getEvents()函数较为复杂，为了给后续分析铺平道路，本节不讨论其细节，先通过伪代码理解此函数的结构与工作方式，在后续深入分析时思路才会比较清晰。

getEvents()函数的本质就是读取并处理Epoll事件与INotify事件。参考以下代码：

```
[EventHub.cpp-->EventHub::getEvents()]
size_t EventHub::getEvents(int timeoutMillis,RawEvent* buffer, size_t bufferSize) {
    /* event指针指向了在buffer下一个可用于存储事件的RawEvent结构体。每存储一个事件，
       event指针都回向后偏移一个元素 */
   RawEvent* event = buffer;
    /*capacity记录了buffer中剩余的元素数量。当capacity为0时，表示buffer已满，此时需要停
       继续处理新事件，并将已处理的事件返回给调用者 */
    size_tcapacity = bufferSize;
    /* 接下来的循环是getEvents()函数的主体。在这个循环中，会先将可用事件放入到buffer中并返回。
       如果没有可用事件，则进入epoll_wait()等待事件的到来，epoll_wait()返回后会重新循环将可用
       将新事件放入buffer */
    for (;;){
       nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
        /* **① 首先进行与设备相关的工作。**某些情况下，如EventHub创建后第一次执行getEvents()函数
           时，需要扫描/dev/input文件夹下的所有设备节点并将这些设备打开。另外，当设备节点的发生增
           动作生时，会将设备事件存入到buffer中 */
        ......
        /* **② 处理未被InputReader取走的输入事件与设备事件。**epoll_wait()所取出的epoll_event
           存储在mPendingEventItems中，mPendingEventCount指定了mPendingEventItems数组
           所存储的事件个数。而mPendingEventIndex指定尚未处理的epoll_event的索引 */
       while (mPendingEventIndex &lt; mPendingEventCount) {
           const struct epoll_event& eventItem =
                                mPendingEventItems[mPendingEventIndex++];
           /* 在这里分析每一个epoll_event，如果是表示设备节点可读，则读取原始事件并放置到buffer
              中。如果是表示mINotifyFd可读，则设置mPendingINotify为true，当InputReader
              将现有的输入事件都取出后读取mINotifyFd中的事件，并进行相应的设备加载与卸载操作。
                另外，如果此epoll_event表示wakeFds的读取端有数据可读，则设置awake标志为true，
              无论此次getEvents()调用有无取到事件，都不会再次进行epoll_wait()进行事件等待 */
           ......
        }
        // ③ 如果mINotifyFd有数据可读，说明设备节点发生了增删操作
        if(mPendingINotify && mPendingEventIndex &gt;= mPendingEventCount) {
           /* 读取mINotifyFd中的事件，同时对输入设备进行相应的加载与卸载操作。这个操作必须当
              InputReader将现有输入事件读取并处理完毕后才能进行，因为现有的输入事件可能来自需要
              被卸载的输入设备，InputReader处理这些事件依赖于对应的设备信息 */
             ......
             deviceChanged= true;
        }
        // 设备节点增删操作发生时，则重新执行循环体，以便将设备变化的事件放入buffer中
        if(deviceChanged) {
           continue;
        }
        // 如果此次getEvents()调用成功获取了一些事件，或者要求唤醒InputReader，则退出循环并
        // 结束getEvents()的调用，使InputReader可以立刻对事件进行处理
        if(event != buffer || awoken) {
           break;
        }
        /* ④ 如果此次getEvents()调用没能获取事件，说明mPendingEventItems中没有事件可用，
           于是执行epoll_wait()函数等待新的事件到来，将结果存储到mPendingEventItems里，并重
           置mPendingEventIndex为0 */
       mPendingEventIndex = 0;
       ......
        intpollResult = epoll_wait(mEpollFd, mPendingEventItems, EPOLL_MAX_EVENTS,timeout
       ......
        mPendingEventCount= size_t(pollResult);
        // 从epoll_wait()中得到新的事件后，重新循环，对新事件进行处理
    }
    // 返回本次getEvents()调用所读取的事件数量
    returnevent - buffer;
}
```

getEvents()函数使用Epoll的核心是mPendingEventItems数组，它是一个事件池。getEvents()函数会优先从这个事件池获取epoll事件进行处理，并将读取相应的原始输入事件返回给调用者。当因为事件池枯竭而导致调用者无法获得任何事件时，会调用epoll_wait()函数等待新事件的到来，将事件池重新注满，然后再重新处理事件池中的Epoll事件。从这个意

义来说，getEvents()函数的调用过程，就是消费epoll_wait()所产生的Epoll事件的过程。因此可以将从epoll_wait()的调用开始，到将Epoll事件消费完毕的过程称为EventHub的一个监听周期。依据每次epoll_wait()产生的Epoll事件的数量以及设备节点中原始输入事件的数量，一个监听周期包含一次或多次getEvents()调用。周期中的第一次调用会因为事件池枯竭而直接进入epoll_wait()，而周期中的最后一次调用一定会将最后的事件取走。

注意 getEvents()采用事件池机制的根本原因是buffer的容量限制。由于一次epoll_wait()可能返回多个设备节点的可读事件，每个设备节点又有可能读取多条原始输入事件，一段时间内原始输入事件的数量可能大于buffer的容量。因此需要一个事件池以缓存因buffer容量不够而无法处理的epoll事件，以便在下次调用时可以将这些事件优先处理。这是缓冲区操作的一个常用技巧。

当有INotify事件可以从mINotifyFd中读取时，会产生一个epoll事件，EventHub便得知设备节点发生了增删操作。在getEvents()将事件池中的所有事件处理完毕后，便会从mINotifyFd中读取INotify事件，进行输入设备的加载/卸载操作，然后生成对应的RawEvent结构体并返回给调用者。

通过上述分析可以看到，getEvents()包含了原始输入事件读取、输入设备加载/卸载等操作。这几乎是EventHub的全部工作了。如果没有geEvents()的调用，EventHub将对输入事件、设备节点增删事件置若罔闻，因此可以将一次getEvents()调用理解为一次心跳，EventHub的核心功能都会在这次心跳中完成。

getEvents()的代码还揭示了另外一个信息：在一个监听周期内的设备增删事件与Epoll事件的优先级。设备事件的生成逻辑位于Epoll事件的处理之前，因此getEvents()将优先生成设备增删事件，完成所有设备增删事件的生成之前不会处理Epoll事件，也就是不会生成原始输入事件。

接下来我们将从设备管理与原始输入事件处理两个方面深入探讨EventHub。

3. 输入设备管理

因为输入设备是输入事件的来源，并且决定了输入事件的含义，因此首先讨论EventHub的输入设备管理机制。

输入设备是一个可以为接收用户操作的硬件，内核会为每一个输入设备在/dev/input/下创建一个设备节点，而当输入设备不可用时（例如被拔出），将其设备节点删除。这个设备节点包含了输入设备的所有信息，包括名称、厂商、设备类型，设备的功能等。除了设备节点，某些输入设备还包含一些自定义配置，这些配置以键值对的形式存储在某个文件中。这些信息决定了Reader子系统如何加工原始输入事件。EventHub负责在设备节点可用时加载并维护这些信息，并在设备节点被删除时将其移除。

EventHub通过一个定义在其内部的名为Device的私有结构体来描述一个输入设备。其定义如下：

```
[EventHub.h-->EventHub::Device]
struct Device {
    Device*next;  /* Device结构体实际上是一个单链表 */
    int fd;          /* fd表示此设备的设备节点的描述符，可以从此描述符中读取原始输入事件 */
    constint32_t id;     /* id在输入系统中唯一标识这个设备，由EventHub在加载设备时进行分配 */
    constString8 path; /* path存储了设备节点在文件系统中的路径 */
    constInputDeviceIdentifier identifier; /* 厂商信息，存储了设备的供应商、型号等信息
                                                         这些信息从设备节点中获得 */
    uint32_tclasses;  /* classes表示了设备的类别，键盘设备，触控设备等。一个设备可以同时属于
                                  多个设备类别。类别决定了InputReader如何加工其原始输入事件 */
    /* 接下来是一系列的事件位掩码，它们详细地描述了设备能够产生的事件类型。设备能够产生的事件类型
       决定了此设备所属的类型*/
    uint8_tkeyBitmask[(KEY_MAX + 1) / 8];
    ......
    /* 配置信息。以键值对的形式存储在一个文件中，其路径取决于identfier字段中的厂商信息，这些
       配置信息将会影响InputReader对此设备的事件的加工行为 */
    String8configurationFile;
   PropertyMap* configuration;
    /* 键盘映射表。对于键盘类型的设备，这些键盘映射表将原始事件中的键盘扫描码转换为Android定义的
      的按键值。这个映射表也是从一个文件中加载的，文件路径取决于dentifier字段中的厂商信息 */
   VirtualKeyMap* virtualKeyMap;
    KeyMapkeyMap;
   sp&lt;KeyCharacterMap&gt; overlayKeyMap;
   sp&lt;KeyCharacterMap&gt; combinedKeyMap;
    // 力反馈相关的信息。有些设备如高级的游戏手柄支持力反馈功能，目前暂不考虑
    boolffEffectPlaying;
    int16_tffEffectId;
};
```

Device结构体所存储的信息主要包括以下几个方面：

- 设备节点信息：保存了输入设备节点的文件描述符、文件路径等。
- 厂商信息：包括供应商、设备型号、名称等信息，这些信息决定了加载配置文件与键盘映射表的路径。
- 设备特性信息：包括设备的类别，可以上报的事件种类等。这些特性信息直接影响了InputReader对其所产生的事件的加工处理方式。
- 设备的配置信息：包括键盘映射表及其他自定义的信息，从特定位置的配置文件中读取。

另外，Device结构体还存储了力反馈所需的一些数据。在本节中暂不讨论。

EventHub用一个名为mDevices的字典保存当前处于打开状态的设备节点的Device结构体。字典的键为设备Id。

（**1**）输入设备的加载

EventHub在创建后在第一次调用getEvents()函数时完成对系统中现有输入设备的加载。

再看一下getEvents()函数中相关内容的实现：

```
[EventHub.cpp-->EventHub::getEvents()]
size_t EventHub::getEvents(int timeoutMillis,RawEvent* buffer, size_t bufferSize) {
    for (;;){
        // 处理输入设备卸载操作
        ......
        /* 在EventHub的构造函数中，mNeedToScanDevices被设置为true，因此创建后第一次调用
          getEvents()函数会执行scanDevicesLocked()，加载所有输入设备 */
        if(mNeedToScanDevices) {
           mNeedToScanDevices = false;
            /***scanDevicesLocked()将会把/dev/input下所有可用的输入设备打开并存储到Device**
**               结构体中** */
           scanDevicesLocked();
           mNeedToSendFinishedDeviceScan = true;
        }
        ......
    }
    returnevent – buffer;
}
```

加载所有输入设备由scanDevicesLocked()函数完成。看一下其实现：

```
[EventHub.cpp-->EventHub::scanDevicesLocked()]
void EventHub::scanDevicesLocked() {
    // 调用scanDirLocked()函数遍历/dev/input文件夹下的所有设备节点并打开
    status_tres = scanDirLocked(DEVICE_PATH);
    ......// 错误处理
    // 打开一个名为VIRTUAL_KEYBOARD的输入设备。这个设备时刻是打开着的。它是一个虚拟的输入设
       备，没有对应的输入节点。读者先记住有这么一个输入设备存在于输入系统中 */
    if(mDevices.indexOfKey(VIRTUAL_KEYBOARD_ID) &lt; 0) {
       createVirtualKeyboardLocked();
    }
}
```

scanDirLocked()遍历指定文件夹下的所有设备节点，分别对其执行openDeviceLocked()完成设备的打开操作。在这个函数中将为设备节点创建并加载Device结构体。参考其代码：

```
[EventHub.cpp-->EventHub::openDeviceLocked()]
status_t EventHub::openDeviceLocked(const char*devicePath) {
    // 打开设备节点的文件描述符，用于获取设备信息以及读取原始输入事件
    int fd =open(devicePath, O_RDWR | O_CLOEXEC);
    // 接下来的代码通过ioctl()函数从设备节点中获取输入设备的厂商信息
   InputDeviceIdentifier identifier;
    ......
    // 分配一个设备Id并创建Device结构体
    int32_tdeviceId = mNextDeviceId++;
    Device*device = new Device(fd, deviceId, String8(devicePath), identifier);
    // 为此设备加载配置信息。、
   loadConfigurationLocked(device);
    // **① 通过ioctl函数获取设备的事件位掩码。**事件位掩码指定了输入设备可以产生何种类型的输入事件
   ioctl(fd, EVIOCGBIT(EV_KEY, sizeof(device->keyBitmask)),device->keyBitmask);
    ......
   ioctl(fd, EVIOCGPROP(sizeof(device->propBitmask)),device->propBitmask);
    // 接下来的一大段内容是根据事件位掩码为设备分配类别，即设置classes字段。、
    ......
    /* **② 将设备节点的描述符的可读事件注册到Epoll中。**当此设备的输入事件到来时，Epoll会在
      getEvents()函数的调用中产生一条epoll事件 */
    structepoll_event eventItem;
   memset(&eventItem, 0, sizeof(eventItem));
   eventItem.events = EPOLLIN;
   eventItem.data.u32 = deviceId; /* 注意，epoll_event的自定义信息是设备的Id
    if(epoll_ctl(mEpollFd, EPOLL_CTL_ADD, fd, &eventItem)) {
        ......
    }
    ......
    // **③ 调用addDeviceLocked()将Device添加到mDevices字典中**
   addDeviceLocked(device);
    return0;
}
```

openDeviceLocked()函数打开指定路径的设备节点，为其创建并填充Device结构体，然后将设备节点的可读事件注册到Epoll中，最后将新建的Device结构体添加到mDevices字典中以供检索之需。整个过程比较清晰，但仍有以下几点需要注意：

- openDeviceLocked()函数从设备节点中获取了设备可能上报的事件类型，并据此为设备分配了类别。整个分配过程非常繁琐，由于它和InputReader的事件加工过程关系紧密，因此这部分内容将在5.2.4节再做详细讨论。
- 向Epoll注册设备节点的可读事件时，epoll_event的自定义数据被设置为设备的Id而不是fd。
- addDeviceLocked()将新建的Device对象添加到mDevices字典中的同时也会将其添加到一个名为mOpeningDevices的链表中。这个链表保存了刚刚被加载，但尚未通过getEvents()函数向InputReader发送DEVICE_ADD事件的设备。

完成输入设备的加载之后，通过getEvents()函数便可以读取到此设备所产生的输入事件了。除了在getEvents()函数中使用scanDevicesLockd()一次性加载所有输入设备，当INotify事件告知有新的输入设备节点被创建时，也会通过opendDeviceLocked()将设备加载，稍后再做讨论。

（**2**）输入设备的卸载

输入设备的卸载由closeDeviceLocked()函数完成。由于此函数的工作内容与openDeviceLocked()函数正好相反，就不列出其代码了。设备的卸载过程为：

- 从Epoll中注销对描述符的监听。
- 关闭设备节点的描述符。
- 从mDevices字典中删除对应的Device对象。
- 将Device对象添加到mClosingDevices链表中，与mOpeningDevices类似，这个链表保存了刚刚被卸载，但尚未通过getEvents()函数向InputReader发送DEVICE_REMOVED事件的设备。

同加载设备一样，在getEvents()函数中有根据需要卸载所有输入设备的操作（比如当EventHub要求重新加载所有设备时，会先将所有设备卸载）。并且当INotify事件告知有设备节点删除时也会调用closeDeviceLocked()将设备卸载。

（**3**）设备增删事件

在分析设备的加载与卸载时发现，新加载的设备与新卸载的设备会被分别放入mOpeningDevices与mClosingDevices链表之中。这两个链表将是在getEvents()函数中向InputReader发送设备增删事件的依据。

参考getEvents()函数的相关代码，以设备卸载事件为例看一下设备增删事件是如何产生的：

```
[EventHub.cpp-->EventHub::getEvents()]
size_t EventHub::getEvents(int timeoutMillis,RawEvent* buffer, size_t bufferSize) {
    for (;;){
        // 遍历mClosingDevices链表，为每一个已卸载的设备生成DEVICE_REMOVED事件
       while (mClosingDevices) {
           Device* device = mClosingDevices;
           mClosingDevices = device->next;
           /* 分析getEvents()函数的工作方式时介绍过，event指针指向buffer中下一个可用于填充
              事件的RawEvent对象 */
           event->when = now; // 设置产生事件的事件戳
           event->deviceId =
                   device->id ==mBuiltInKeyboardId ? BUILT_IN_KEYBOARD_ID : device->id;
           event->type = DEVICE_REMOVED; // 设置事件的类型为DEVICE_REMOVED
           event += 1; // 将event指针移动到下一个可用于填充事件的RawEvent对象
           delete device; // 生成DEVICE_REMOVED事件之后，被卸载的Device对象就不再需要了
           mNeedToSendFinishedDeviceScan = true; // 随后发送FINISHED_DEVICE_SCAN事件
            /* 当buffer已满则停止继续生成事件，将已生成的事件返回给调用者。尚未生成的事件
              将在下次getEvents()调用时生成并返回给调用者 */
           if (--capacity == 0) {
               break;
           }
        }
        // 接下来进行DEVICE_ADDED事件的生成，此过程与 DEVICE_REMOVED事件的生成一致
       ......
    }
    returnevent – buffer;
}
```

可以看到，在一次getEvents()调用中会尝试为所有尚未发送增删事件的输入设备生成对应的事件返回给调用者。表示设备增删事件的RawEvent对象包含三个信息：产生事件的事件戳、产生事件的设备Id，以及事件类型（DEVICE_ADDED或DEVICE_REMOVED）。

当生成设备增删事件时，会设置mNeedToSendFinishedDeviceSan为true，这个动作的意思是完成所有DEVICE_ADDED/REMOVED事件的生成之后，需要向getEvents()的调用者发送一个FINISHED_DEVICE_SCAN事件，表示设备增删事件的上报结束。这个事件仅包括时间戳与事件类型两个信息。

经过以上分析可知，EventHub可以产生的设备增删事件一共有三种，而且这三种事件拥有固定的优先级，DEVICE_REMOVED事件的优先级最高，DEVICE_ADDED事件次之，FINISHED_DEVICE_SCAN事件最低。而且，getEvents()完成当前高优先级事件的生成之前，不会进行低优先级事件的生成。因此，当发生设备的加载与卸载时，EventHub所生成的完整的设备增删事件序列如图5-5所示，其中R表示DEVICE_REMOVED，A表示DEVICE_ADDED，F表示FINISHED_DEVICE_SCAN。



图 5-5 设备增删事件的完整序列

由于参数buffer的容量限制，这个事件序列可能需要通过多次getEvents()调用才能完整地返回给调用者。另外，根据5.2.2节的讨论，设备增删事件相对于Epoll事件拥有较高的优先级，因此从R1事件开始生成到F事件生成之前，getEvents()不会处理Epoll事件，也就是说不会生成原始输入事件。

总结一下设备增删事件的生成原理：

- 当发生设备增删时，addDeviceLocked()函数与closeDeviceLocked()函数会将相应的设备放入mOpeningDevices和mClosingDevices链表中。
- getEvents()函数会根据mOpeningDevices和mClosingDevices两个链表生成对应DEVICE_ADDED和DEVICE_REMOVED事件，其中后者的生成拥有高优先级。
- DEVICE_ADDED和DEVICE_REMOVED事件都生成完毕后，getEvents()会生成FINISHED_DEVICE_SCAN事件，标志设备增删事件序列的结束。

（4）通过INotify动态地加载与卸载设备

通过前文的介绍知道了openDeviceLocked()和closeDeviceLocked()可以加载与卸载输入设备。接下来分析EventHub如何通过INotify进行设备的动态加载与卸载。在EventHub的构造函数中创建了一个名为mINotifyFd的INotify对象的描述符，用以监控/dev/input下设备节点的增删。之后将mINotifyFd的可读事件加入到Epoll中。于是可以确定动态加载与卸载设备的工作

方式为：首先筛选epoll_wait()函数所取得的Epoll事件，如果Epoll事件表示了mINotifyFd可读，便从mINotifyFd中读取设备节点的增删事件，然后通过执行openDeviceLocked()或closeDeviceLocked()进行设备的加载与卸载。

看一下getEvents()中与INotify相关的代码：

```
[EventHub.cpp-->EventHub::getEvents()]
size_t EventHub::getEvents(int timeoutMillis,RawEvent* buffer, size_t bufferSize) {
    for (;;){
       ...... // 设备增删事件处理
        while(mPendingEventIndex &lt; mPendingEventCount) {
           const struct epoll_event& eventItem =
                                mPendingEventItems[mPendingEventIndex++];
           /* **① 通过Epoll事件的data字段确定此事件表示了mINotifyFd可读**
              注意EPOLL_ID_INOTIFY在EventHub的构造函数中作为data字段向
              Epoll注册mINotifyFd的可读事件 */
           if (eventItem.data.u32 == EPOLL_ID_INOTIFY) {
               if (eventItem.events & EPOLLIN) {
                   mPendingINotify = true; // 标记INotify事件待处理
               } else { ...... }
               continue; // 继续处理下一条Epoll事件
           }
           ...... // 其他Epoll事件的处理
        }
        // 如果INotify事件待处理
        if(mPendingINotify && mPendingEventIndex &gt;= mPendingEventCount) {
           mPendingINotify = false;
           /* **② 调用readNotifyLocked()函数读取并处理存储在mINotifyFd中的INotify事件**
              这个函数将完成设备的加载与卸载 */
           readNotifyLocked();
           deviceChanged = true;
        }
        //**③ 如果处理了INotify事件，则返回到循环开始处，生成设备增删事件**
        if(deviceChanged) {
           continue;
        }
    }
}
```

getEvents()函数中与INotify相关的代码共有三处：

- 识别表示mINotifyFd可读的Epoll事件，并通过设置mPendingINotify为true以标记有INotify事件待处理。getEvents()并没有立刻处理INotify事件，因为此时进行设备的加载与卸载是不安全的。其他Epoll事件可能包含了来自即将被卸载的设备的输入事件，因此需要将所有Epoll事件都处理完毕后再进行加载与卸载操作。
- 当epoll_wait()所返回的Epoll事件都处理完毕后，调用readNotifyLocked()函数读取mINotifyFd中的事件，并进行设备的加载与卸载操作。
- 完成设备的动态加载与卸载后，需要返回到循环最开始处，以便设备增删事件处理代码生成设备的增删事件。

其中第一部分与第三部分比较容易理解。接下来看一下readNotifyLocked()是如何工作的。

```
[EventHub.cpp-->EventHub::readNotifyLocked()]
status_t EventHub::readNotifyLocked() {
    ......
    // 从mINotifyFd中读取INotify事件列表
    res =read(mINotifyFd, event_buf, sizeof(event_buf));
    ......
    // 逐个处理列表中的事件
   while(res &gt;= (int)sizeof(*event)) {
       strcpy(filename, event->name); // 从事件中获取设备节点路径
       if(event->mask & IN_CREATE) {
           openDeviceLocked(devname); // 如果事件类型为IN_CREATE，则加载对应设备
        }else {
           closeDeviceByPathLocked(devname); // 否则卸载对应设备
        }
        ......// 移动到列表中的下一个事件
    }
    return0;
}
```

（5） **EventHub设备管理总结**

至此，EventHub的设备管理相关的知识便讨论完毕了。在这里进行一下总结：

- EventHub通过Device结构体描述输入设备的各种信息。
- EventHub在getEvents()函数中进行设备的加载与卸载操作。设备的加载与卸载分为按需加载或卸载以及通过INotify动态加载或卸载特定设备两种方式。
- getEvents()函数进行了设备的加载与卸载操作后，会生成DEVICE_ADDED、DEVICE_REMOVED以及FINISHED_DEVICE_SCAN三种设备增删事件，并且设备增删事件拥有高于Epoll事件的优先级。

4. 原始输入事件的监听与读取

本节将讨论EventHub另一个核心的功能，监听与读取原始输入事件。

回忆一下输入设备的加载过程，当设备加载时，openDeviceLocked()会打开设备节点的文件描述符，并将其可读事件注册进Epoll中。于是当设备的原始输入事件到来时，getEvents()函数将会获得一条Epoll事件，然后根据此Epoll事件读取文件描述符中的原始输入事件，将其填充到RawEvents结构体并放入buffer中被调用者取走。openDeviceLocked()注册了设备节点的EPOLLIN和EPOLLHUP两个事件，分别表示可读与被挂起（不可用），因此getEvents()需要分别处理这两种事件。

看一下getEvents()函数中的相关代码：

```
[EventHub.cpp-->EventHub::getEvents()]
size_t EventHub::getEvents(int timeoutMillis,RawEvent* buffer, size_t bufferSize) {
    for (;;){
       ...... // 设备增删事件处理
        while(mPendingEventIndex &lt; mPendingEventCount) {
           const struct epoll_event& eventItem =
                                  mPendingEventItems[mPendingEventIndex++];
           ...... // INotify与wakeFd的Epoll事件处理
           /* **① 通过Epoll的data.u32字段获取设备Id，进而获取对应的Device对象。**如果无法找到
              对应的Device对象，说明此Epoll事件并不表示原始输入事件的到来，忽略之 */
           ssize_t deviceIndex = mDevices.indexOfKey(eventItem.data.u32);
           Device* device = mDevices.valueAt(deviceIndex);
           ......
           if (eventItem.events & EPOLLIN) {
               /* **② 如果Epoll事件为EPOLLIN，表示设备节点有原始输入事件可读**。此时可以从描述符
                  中读取。读取结果作为input_event结构体并存储在readBuffer中，注意事件的个数
                   受到capacity的限制*/
               int32_t readSize = read(device->fd, readBuffer,
                        sizeof(structinput_event) * capacity);
               if (......) {                    ......// 一些错误处理 }
                else {
                   size_t count = size_t(readSize) / sizeof(struct input_event);
                   /* **② 将读取到的每一个input_event结构体中的数据转换为一个RawEvent对象，**
                       并存储在buffer参数中以返回给调用者 */
                   for (size_t i = 0; i &lt; count; i++) {
                        const structinput_event& iev = readBuffer[i];
                        ......
                        event->when = now;
                        event->deviceId =deviceId;
                        event->type =iev.type;
                        event->code =iev.code;
                        event->value =iev.value;
                       event += 1; // 移动到buffer的下一个可用元素
                   }
                   /* 接下来的一个细节需要注意，因为buffer的容量限制，可能无法完全读取设备节点
                       中存储的原始事件。一旦buffer满了则需要立刻返回给调用者。设备节点中剩余的
                       输入事件将在下次getEvents()调用时继续读取，也就是说，当前的Epoll事件
                       并未处理完毕。mPendingEventIndex -= 1的目的就是使下次getEvents()调用
                       能够继续处理这个Epoll事件 */
                   capacity -= count;
                   if (capacity == 0) {
                        mPendingEventIndex -=1;
                        break;
                   }
               }
           } else if (eventItem.events & EPOLLHUP) {
               deviceChanged = true; // 如果设备节点的文件描述符被挂起则卸载此设备
               closeDeviceLocked(device);
           } else { ...... }
        }
       ...... // 读取并处理INotify事件
        ......// 等待新的Epoll事件
    }
    returnevent – buffer；
}
```

getEvents()通过Epoll事件的data.u32字段在mDevices列表中查找已加载的设备，并从设备的文件描述符中读取原始输入事件列表。从文件描述符中读取的原始输入事件存储在input_event结构体中，这个结构体的四个字段存储了事件的事件戳、类型、代码与值四个元素。然后逐一将input_event的数据转存到RawEvent中并保存至buffer以返回给调用者。

注意 为了叙述简单，上述代码使用了调用getEvents()的时间作为输入事件的时间戳。由于调用getEvents()函数的时机与用户操作的时间差的存在，会使得此时间戳与事件的真实时间有所偏差。从设备节点中读取的input_event中也包含了一个时间戳，这个时间戳消除了getEvents()调用所带来的时间差，因此可以获得更精确的时间控制。可以通过打开HAVE_POSIX_CLOCKS宏以使用input_event中的时间而不是将getEvents()调用的时间作为输入事件的时间戳。

需要注意的是，由于Epoll事件的处理优先级低于设备增删事件，因此当发生设备加载与卸载动作时，不会产生设备输入事件。另外还需注意，在一个监听周期中，getEvents()在将一个设备节点中的所有原始输入事件读取完毕之前，不会读取其他设备节点中的事件。

5. EventHub总结

本节针对EventHub的设备管理与原始输入事件的监听读取两个核心内容介绍了EventHub的工作原理。EventHub作为直接操作设备节点的输入系统组件，隐藏了INotify与Epoll以及设备节点读取等底层操作，通过一个简单的接口getEvents()向使用者提供抽取设备事件与原始输入事件的功能。EventHub的核心功能都在getEvents()函数中完成，因此深入理解getEvents()的工作原理对于深入理解EventHub至关重要。

getEvents()函数的本质是通过epoll_wait()获取Epoll事件到事件池，并对事件池中的事件进行消费的过程。从epoll_wait()的调用开始到事件池中最后一个事件被消费完毕的过程称之为EventHub的一个监听周期。由于buffer参数的尺寸限制，一个监听周期可能包含多个getEvents()调用。周期中的第一个getEvents()调用一定会因事件池的枯竭而直接进行epoll_wait()，而周期中的最后一个getEvents()一定会将事件池中的最后一条事件消费完毕并将事件返回给调用者。前文所讨论的事件优先级都是在同一个监听周期内而言的。

在本节中出现了很多种事件，有原始输入事件、设备增删事件、Epoll事件、INotify事件等，存储事件的结构体有RawEvent、epoll_event、inotify_event、input_event等。图5-6可以帮助读者理清这些事件之间的关系。

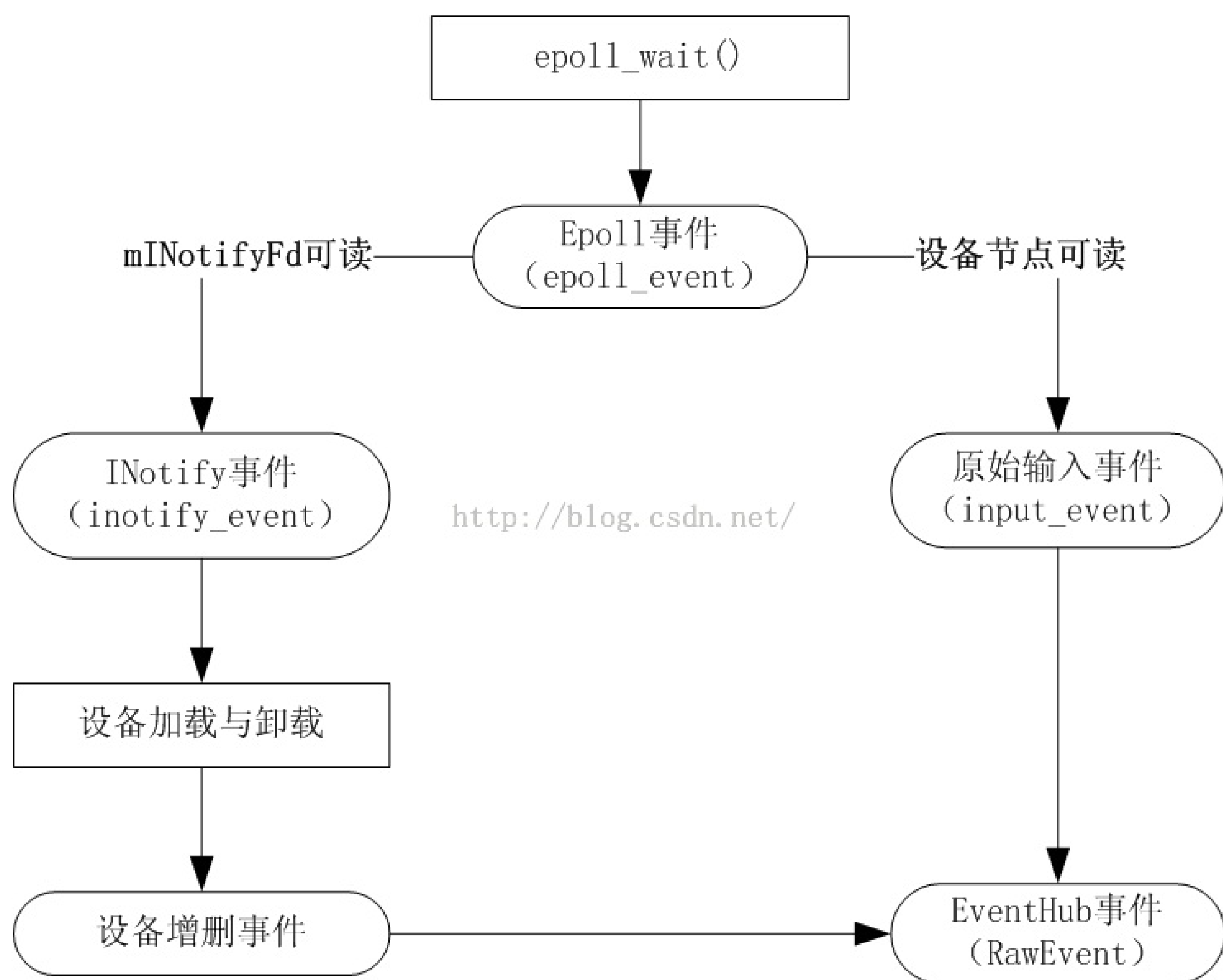


图 5-6 EventHub的事件关联

另外，getEvents()函数返回的事件列表依照事件的优先级拥有特定的顺序。并且在一个监听周期中，同一输入设备的输入事件在列表中是相邻的。

至此，相信读者对EventHub的工作原理，以及EventHub的事件监听与读取机制有了深入的了解。接下来的内容将讨论EventHub所提供的原始输入事件如何被加工为Android输入事件，这个加工者就是Reader子系统中的另一员大将：InputReader。

> [1] 感兴趣的读者可以通过`gitclone git://github.com/barzooka/robert.git`下载一个可以录制用户输入操作并可以实时回放的小工具。

第6章 深入理解控件（ViewRoot）系统（节选）

本章主要内容：

- 介绍创建窗口的新的方法以及WindowManager的实现原理
- 探讨ViewRootImpl的工作方式
- 讨论控件树的测量、布局与绘制
- 讨论输入事件在控件树中的派发
- 介绍PhoneWindow的工作原理以及Activity窗口的创建方式

本章涉及的源代码文件名及位置：

- ContextImpl.java

frameworks/base/core/java/android/app/ContextImpl.java

- WindowManagerImpl.java

frameworks/base/core/java/android/view/WindowManagerImpl.java

- WindowManagerGlobal.java

frameworks/base/core/java/android/view/WindowManagerGlobal.java

- ViewRootImpl.java

frameworks/base/core/java/android/view/ViewRootImpl.java

- View.java

frameworks/base/core/java/android/view/View.java

- ViewGroup.java

frameworks/base/core/java/android/view/ViewGroup.java

- TabWidget.java

frameworks/base/core/java/android/widget/TabWidget.java

- HardwareRenderer.java

frameworks/base/core/java/android/view/HardwareRenderer.java

- FocusFinder.java

frameworks/base/core/java/android/view/FocusFinder.java

- Activity.java

frameworks/base/core/java/android/app/Activity.java

- PhoneWindow.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java

- Window.java

frameworks/base/core/java/android/view/Window.java

- ActivityThread.java

frameworks/base/core/java/android/app/ActivityThread.java

6.1 初识Android的控件系统

第4章和第5章分别介绍了窗口的两个最核心的内容：显示与用户输入，同时也介绍了在Android中显示一个窗口并接受输入事件的最基本的方法。但是这种方法过于基本，不便于使用。直接使用Canvas绘制用户界面以及使用InputEventReceiver处理用户输入是一件非常繁琐恼人的工作，因为你不得不亲历亲为以下复杂的工作：

- 测量各个UI元素（一段文字、一个图片）的显示尺寸与位置。
- 对各个UI元素进行布局计算与绘制。
- 当显示内容需要发生变化时进行重绘。出于效率考虑，你必须保证重绘区域尽可能地小。
- 分析InputEventReceiver所接收的事件的类型，并确定应该由哪个UI元素响应这个事件。
- 需要处理来自WMS的很多与窗口状态相关的回调。

所幸Android的控件系统使得这些事情不需要我们亲历亲为。

自1983年苹果公司发布第一款搭载图形用户界面(GUI)操作系统的个人电脑Lisa以来的三十多年里，图形用户界面已经发展得相当成熟。无论是运行于桌面系统还是Web，每一个面向图形用户界面的开发工具包(SDK)都至少内置实现了用户和开发者所公认的一套UI元素，尽管名称可能有所差异。例如文本框、图片框、列表框、组合框、按钮、单选按钮、多选按钮，等等。Android的控件系统不仅延续了对各种标准UI元素的支持，还针对移动平台的操作特点增加了使用更加方便、种类更加丰富的一系列新型的UI元素。

注意 在Android中，一个UI元素被称为一个视图（View），然而，笔者认为控件才是UI元素的更贴切的名字。因为UI元素不仅仅是为了向用户显示一些内容，更重要的是它们响应用户的输入并进行相应的工作。本书后续部分将以控件来称呼UI元素(View)。

另外，本章的目的并不是介绍如何使用各种Android控件，而是介绍Android控件系统的工作原理。本章要求读者至少应了解使用Android控件的基本知识。

读者所熟知的Activity、各种对话框、弹出菜单、状态栏与导航栏等等都是基于这套控件系统实现的。因此控件系统将是继WMS与IMS两大系统服务之后的又一个需要我们攻克的目标。

6.1.1 另一种创建窗口的方法

在这一小节里将介绍另外一种创建窗口的方法，并以此为切入点来开始对Android控件系统的探讨。

这个例子将会在屏幕中央显示一个按钮，它会浮在所有应用之上，直到用户点击它为止。市面上某些应用的悬浮窗就是如此实现的。

- 首先，读者使用Eclipse建立一个新的Android工程，并新建一个Service。然后在这个Service中增加如下代码：

```
// 将按钮作为一个窗口添加到WMS中
private void installFloatingWindow() {
    // ① 获取一个WindowManager实例
    finalWindowManager wm =
                    (WindowManager)getSystemService(Context.WINDOW_SERVICE);
    // ② 新建一个按钮控件
    finalButton btn = new Button(this.getBaseContext());
   btn.setText("Click me to dismiss!");
    // ③ 生成一个WindowManager.LayoutParams，用以描述窗口的类型与位置信息
   LayoutParams lp = createLayoutParams();
    // ④ 通过WindowManager.addView()方法将按钮作为一个窗口添加到系统中
   wm.addView(btn, lp);
   btn.setOnClickListener(new View.OnClickListener() {
       @Override
       public void onClick(View v) {
            // ⑤当用户点击按钮时，将按钮从系统中删除
           wm.removeView(btn);
           stopSelf();
        }
    });
}
    privateLayoutParams createLayoutParams() {
       LayoutParams lp = new WindowManager.LayoutParams();
       lp.type = LayoutParams.TYPE_PHONE;
       lp.gravity = Gravity.CENTER;
       lp.width = LayoutParams.WRAP_CONTENT;
       lp.height = LayoutParams.WRAP_CONTENT;
       lp.flags = LayoutParams.FLAG_NOT_FOCUSABLE
               | LayoutParams.FLAG_NOT_TOUCH_MODAL;
       return lp;
    }
```

- 然后在新建的Service的onStartCommand()函数中增加对installFloatingWindow()的调用。

- 在应用程序的主Activity的onCreate()函数中调用startService()以启动这个服务。
- 在应用程序的AndroidManifest.xml中增加对权限android.permission.SYSTEM_ALERT_WINDOW的使用声明。

当完成这些工作之后，运行这个应用即可得到如图6-1所示的效果。一个名为“Clickme to dismiss!”的按钮浮在其他应用之上。而点击这个按钮后，它便消失了。

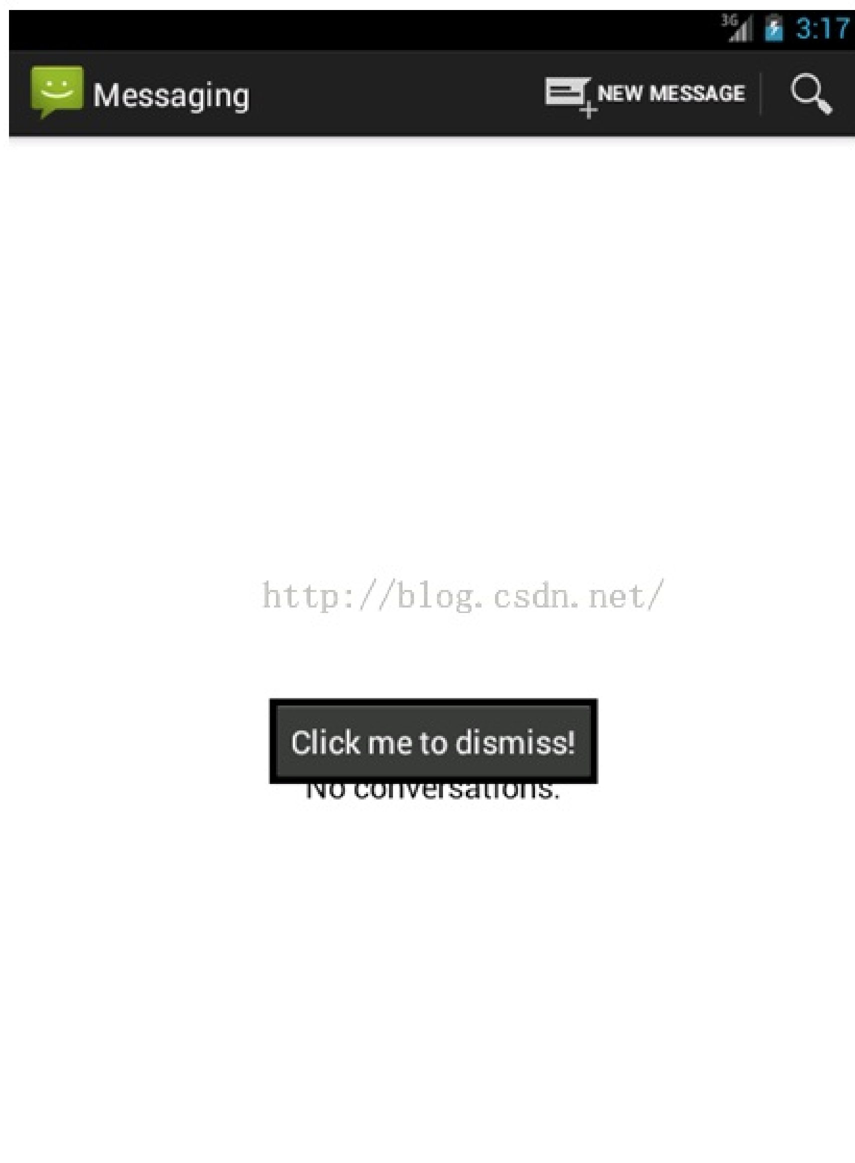


图 6 - 1浮动窗口例子的运行效果

读者可以将本例与第4章的例子SampleWindow做一个对比。它们的实现效果是大同小异的。而然，本章的这个例子无论是从最终效果、代码量、API的复杂度或可读性上都有很大的优势。这得益于对控件系统的使用。在这里，控件Button托管了窗口的绘制过程，并且将输入事件封装为了更具可读性的回调。并且添加窗口时所使用的WindowManager实例掩盖了客户端与WMS交互的复杂性。更重要的是，本例所使用的接口都来自公开的API，也就是说可以脱离Android源码进行编译。这无疑会带来更方便的开发过程以及更好的程序兼容性。

因此，除非需要进行很底层的窗口控制，使用本例所介绍的方法向系统中添加窗口是最优的选择。

6.1.2 控件系统的组成

从这个例子中可以看到在添加窗口过程中的两个关键组件：Button和WindowManager。Button是控件的一种，继承自View类。不只Button，任何一个继承自View类的控件都可以作为一个窗口添加到系统中去。WindowManager其实是一个继承自ViewManager的接口，它提供了添加/删除窗口，更新窗口布局的API，可以看作是WMS在客户端的代理类。不过WindowManager的接口与WMS的接口相差很大，几乎已经无法通过WindowManager看到WMS的模样。这也说明了WindowManager为了精简WMS的接口做过大量的工作。这部分内容也是本章的重点。

因此控件系统便可以分为继承自View类的一系列控件类与WindowManager两个部分。

6.2 深入理解WindowManager

WindowManager的主要功能是提供简单的API使得使用者可以方便地将一个控件作为一个窗口添加到系统中。本节将探讨它工作原理。

6.2.1 WindowManager的创建与体系结构

首先需要搞清楚WindowManager是什么。

准确的说，WindowManager是一个继承自ViewManager的接口。ViewManager定义了三个函数，分别用于添加/删除一个控件，以及更新控件的布局。

ViewManager接口的另一个实现者是ViewGroup，它是容器类控件的基类，用于将一组控件容纳到自身的区域中，这一组控件被称为子控件。ViewGroup可以根据子控件的布局参数（LayoutParams）在其自身的区域中对子控件进行布局。

读者可以将WindowManager与ViewGroup进行一下类比：设想WindowManager是一个ViewGroup，其区域为整个屏幕，而其中的各个窗口就是一个一个的View。WindowManager通过WMS的帮助将这些View按照其布局参数（LayoutParams）将其显示到屏幕的特定位置。二者的核心工作是一样的，因此WindowManager与ViewGroup都继承自ViewManager。

接下来看一下WindowManager接口的实现者。本章最开始的例子通过Context.getSystemService(Context.WINDOW_SERVICE)的方式获取了一个WindowManager的实例，其实现如下：

```
[ContextImpl.java-->ContextImpl.getSystemService()]
public Object getSystemService(String name) {
    // 获取WINDOW_SERVICE所对应的ServiceFetcher
   ServiceFetcher fetcher = SYSTEM_SERVICE_MAP.get(name);
    // 调用fetcher.getService()获取一个实例
    returnfetcher == null ? null : fetcher.getService(this);
}
```

Context的实现者ContextImpl在其静态构造函数中初始化了一系列的ServiceFetcher来响应getSystemService()的调用并创建对应的服务实例。看一下WINDOW_SERVICE所对应的ServiceFetcher的实现：

```
[ContextImpl.java-->ContextImpl.static()]
registerService(WINDOW_SERVICE, newServiceFetcher() {
           public Object getService(ContextImpl ctx) {
               // ① 获取Context中所保存的Display对象
               Display display = ctx.mDisplay;
               /* ② 倘若Context中没有保存任何Display对象，则通过DisplayManager获取系统
                  **主屏幕所对应的Display对象** */
               if (display == null) {
                   DisplayManager dm =
                            (DisplayManager)ctx.getOuterContext().getSystemService(
                                                                Context.DISPLAY_SERVIC
                   display = dm.getDisplay(Display.DEFAULT_DISPLAY);
               }
               // ③ 使用Display对象作为构造函数创建一个WindowManagerImpl对象并返回
               return new WindowManagerImpl(display);
           }});
```

由此可见，通过Context.getSystemService()的方式获取的WindowManager其实是WindowManagerImpl类的一个实例。这个实例的构造依赖于一个Display对象。第4章介绍过DisplayContent的概念，它在WMS中表示一块的屏幕。而这里的Display对象与DisplayContent的意义是一样的，也用来表示一块屏幕。

再看一下WindowManagerImpl的构造函数：

```
[WindowManagerImpl.java-->WindowManagerImpl.WindowManagerImpl()]
    publicWindowManagerImpl(Display display) {
       this(display, null);
    }
    privateWindowManagerImpl(Display display, Window parentWindow) {
       mDisplay = display;
       mParentWindow = parentWindow;
    }
```

其构造函数实在是出奇的简单，仅仅初始化了mDisplay与mParentWindow两个成员变量而已。从这两个成员变量的名字与类型来推断，它们将决定通过这个WindowManagerImpl实例所添加的窗口的归属。

说明 WindowManagerImpl的构造函数引入了一个Window类型的参数parentWindow。Window类是什么呢？以Activity为例，一个Activity显示在屏幕上时包含了标题栏、菜单按钮等控件，但是在setContentView()时并没有在layout中放置它们。这是因为Window类预先为我们

准备好了这一切，它们被称之为窗口装饰。除了产生窗口装饰之外，Window类还保存了窗口相关的一些重要信息。例如窗口ID（IWindow.asBinder()的返回值）以及窗口所属Activity的ID(即AppToken)。在6.6.1 介将会对这个类做详细的介绍。

也许在WindowManagerImpl的addView()函数的实现中可以找到更多的信息。

```
[WindowManagerImpl.java-->WindowManagerImpl.addView()]
    publicvoid addView(View view, ViewGroup.LayoutParams params) {
       mGlobal.addView(view, params, mDisplay, mParentWindow);
    }
```

WindowManagerImpl.addView()将实际的操作委托给一个名为mGlobal的成员来完成，它随着WindowManagerImpl的创建而被初始化：

```
    privatefinal WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();
```

可见mGlobal的类型是WindowManagerGlobal，而且WindowManagerGlobal是一个单例模式——即一个进程中最多仅有一个WindowManagerGlobal实例。所有WindowManagerImpl都是这个进程唯一的WindowManagerGlobal实例的代理。

此时便对WindowManager的结构体系有了一个清晰的认识，如图6-2所示。

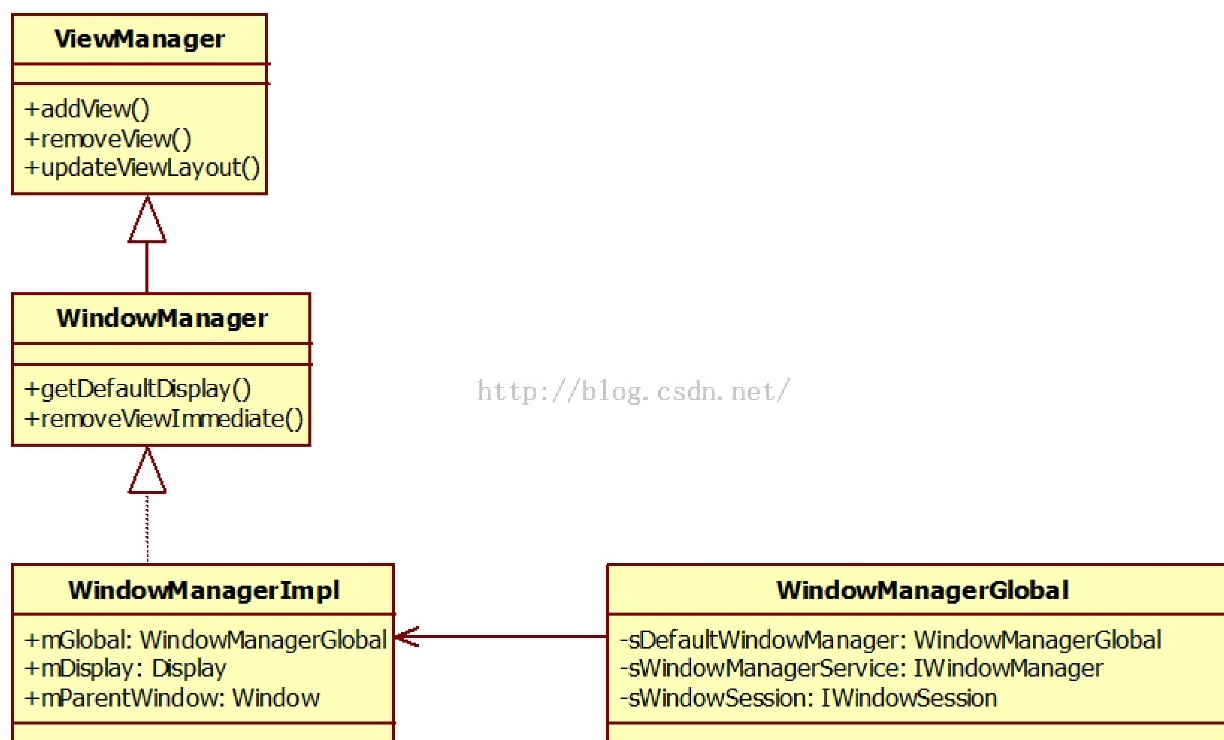


图 6 - 2 WindowManager的结构体系

- ViewManager接口：WindowManager体系中最基本的接口。WindowManager继承自这个接口说明了WindowManager与ViewGroup本质上的一致性。

- WindowManager接口：WindowManager接口继承自ViewManager接口的同时，根据窗口的一些特殊性增加了两个新的接口。getDefaultDisplay()用以得知这个WindowManager的实例会将窗口添加到哪个屏幕上去。而removeViewImmediate()则要求WindowManager必须在这个调用返回之前完成所有的销毁工作。
- WindowManagerImpl类：WindowManager接口的实现者。它自身没有什么实际的逻辑，WindowManager所定义的接口都是交由WindowManagerGlobal完成的。但是它保存了两个重要的只读成员，它们分别指明了通过这个WindowManagerImpl实例所管理的窗口将被显示在哪个屏幕上，以及将会作为哪个窗口的子窗口。因此在一个进程中，WindowManagerImpl的实例可能有多个。
- WindowManagerGlobal类：它没有继承上述任何一个接口，但它是WindowManager的最终实现者。它维护了当前进程中所有已经添加到系统中的窗口的信息。另外，在一个进程中仅有一个WindowManagerGlobal的实例。

在理清了WindowManager的结构体系后，便可以探讨WindowManager是如何完成窗口管理的。其管理方式体现在其对ViewManager的三个接口的实现上。为了简洁起见，我们将直接分析WindowManagerGlobal中的实现。

6.2.2 通过WindowManagerGlobal添加窗口

参考WindowManagerGlobal.addView()的代码：

```
[WindowManagerGlobal.java-->WindowManagerGlobal.addView()]
    publicvoid addView(View view, ViewGroup.LayoutParams params,
            Display display, Window parentWindow){
        ......// 参数检查
       final WindowManager.LayoutParams wparams =(WindowManager.LayoutParams)params;
        /* ① 如果当前窗口需要被添加为另一个窗口的附属窗口（子窗口），则需要让父窗口视自己的情况
            对当前窗口的布局参数(LayoutParams)进行一些修改 */
        if(parentWindow != null) {
           parentWindow.adjustLayoutParamsForSubWindow(wparams);
        }
       ViewRootImpl root;
        ViewpanelParentView = null;
       synchronized (mLock) {
            ......
           // WindowManager不允许同一个View被添加两次
           int index = findViewLocked(view, false);
           if (index &gt;= 0) { throw new IllegalStateException("......");}
           // ② 创建一个ViewRootImpl对象并保存在root变量中
           root = new ViewRootImpl(view.getContext(), display);
           view.setLayoutParams(wparams);
           /* ③ 将作为窗口的控件、布局参数以及新建的ViewRootImpl以相同的索引值保存在三个
              **数组中。**到这步为止，我们可以认为完成了窗口信息的添加工作 */
           mViews[index] = view;
            mRoots[index] = root;
           mParams[index] = wparams;
        }
        try{
           /* **④ 将作为窗口的控件设置给ViewRootImpl。**这个动作将导致ViewRootImpl向WMS
               添加新的窗口、申请Surface以及托管控件在Surface上的重绘动作。这才是真正意义上
                完成了窗口的添加操作*/
           root.setView(view, wparams, panelParentView);
        }catch (RuntimeException e) { ...... }
    }
```

添加窗口的代码并不复杂。其中的关键点有：

- 父窗口修改新窗口的布局参数。可能修改的只有LayoutParams.token和LayoutParams.mTitle两个属性。mTitle属性不必赘述，仅用于调试。而token属性则值得一提。回顾一下第4章的内容，每一个新窗口必须通过LayoutParams.token向WMS出示相应的令牌才可以。在addView()函数中通过父窗口修改这个token属性的目的是为了减少开发者的负担。开发者不需要关心token到底应该被设置为什么值，只需将LayoutParams丢给一个WindowManager，剩下的事情就不用再关心了。父窗口修改token属性的原则是：如果新窗口的类型为子窗口(其类型大于等于LayoutParams.FIRST_SUB_WINDOW并小于等于LayoutParams.LAST_SUB_WINDOW)，则LayoutParams.token所持有的令牌为其父窗口的ID（也就是IWindow.asBinder()的返回值）。否则LayoutParams.token将被修改为父窗口所属的Activity的ID(也就是在第4章中所介绍的AppToken)，这对类型为TYPE_APPLICATION的新窗口来说非常重要。从这点来说，当且仅当新窗的类型为子窗口时addView()的parentWindow参数才是真正意义上的父窗口。这类子窗口有上下文菜单、弹出式菜单以及游标等等，在WMS中，这些窗口对应的WindowState所保存的mAttachedWindow既是parentWindow所对应的WindowState。然而另外还有一些窗口，如对话框窗口，类型为TYPE_APPLICATION，并不属于子窗口，但需要AppToken作为其令牌，为此parentWindow将自己的AppToken赋予了新窗口的的LayoutParams.token中。此时parentWindow便并不是严格意义上的父窗口了。

- 为新窗口创建一个ViewRootImpl对象。顾名思义，ViewRootImpl实现了一个控件树的根。它负责与WMS进行直接的通讯，负责管理Surface，负责触发控件的测量与布局，负责触发控件的绘制，同时也是输入事件的中转站。总之，ViewRootImpl是整个控件系统正常运转的动力所在，无疑是本章最关键的一个组件。

- 将控件、布局参数以及新建的ViewRootImpl以相同的索引值添加到三个对应的数组mViews、mParams以及mRoots中，以供之后的查询之需。控件、布局参数以及ViewRootImpl三者共同组成了客户端的一个窗口。或者说，在控件系统中的窗口就是控件、布局参数与ViewRootImpl对象的一个三元组。

注意 笔者并不认同将这个三元组分别存储在三个数组中的设计。如果创建一个WindowRecord类来统一保存这个三元组将可以省去很多麻烦。

另外，mViews、mParams以及mRoots这三个数组的容量是随着当前进程中的窗口数量的变化而变化的。因此在addView()以及随后的removeView()中都伴随着数组的新建、拷贝等操作。鉴于一个进程所添加的窗口数量不会太多，而且也不会很频繁，所以这些时间开销是可以接受的。不过笔者仍然认为相对于数组，ArrayList或CopyOnWriteArrayList是更好的选择。

- 调用ViewRootImpl.setView()函数，将控件交给ViewRootImpl进行托管。这个动作将使得ViewRootImpl向WMS添加窗口、获取Surface以及重绘等一系列的操作。这一步是控件能够作为一个窗口显示在屏幕上的根本原因！

总体来说，WindowManagerGlobal在通过父窗口调整了布局参数之后，将新建的ViewRootImpl、控件以及布局参数保存在自己的三个数组中，然后将控件交由新建的ViewRootImpl进行托管，从而完成了窗口的添加。WindowManagerGlobal管理窗口的原理如图6-3所示。

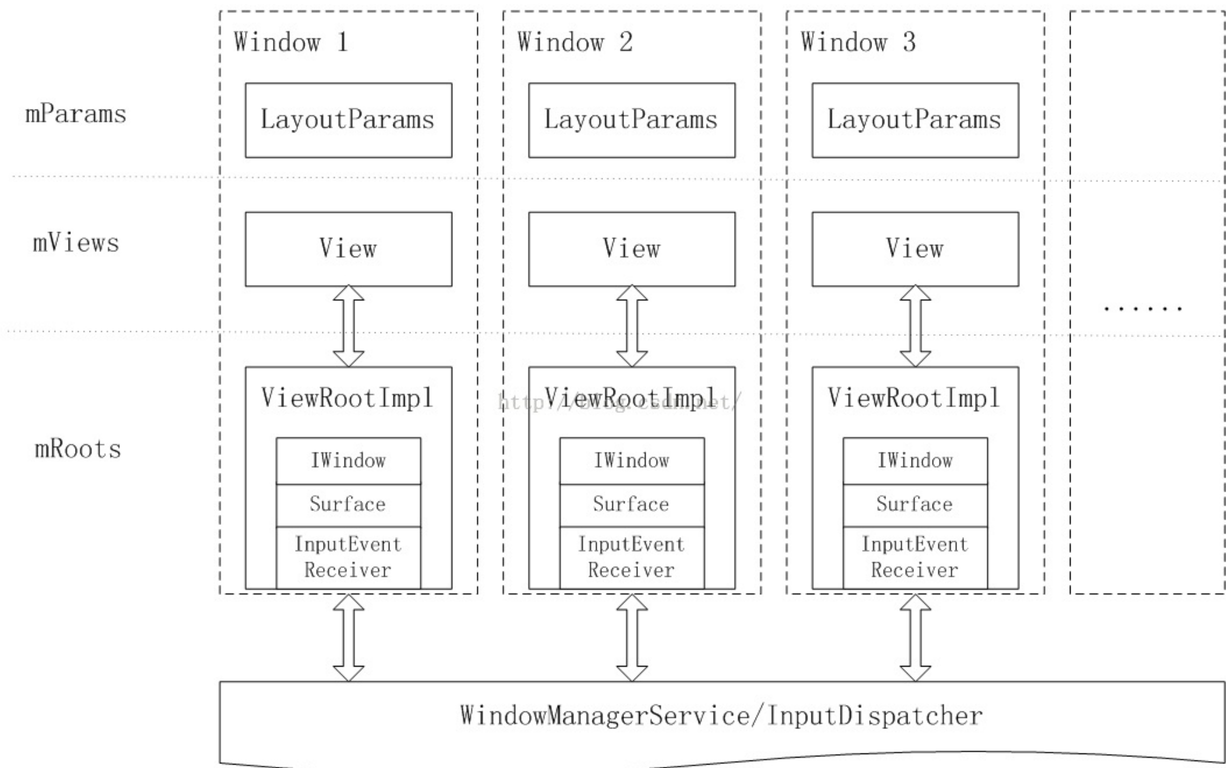


图 6 - 3 WindowManagerGlobal的窗口管理

6.2.3 更新窗口的布局

ViewManager所定义的另外一个功能就是更新View的布局。在WindowManager中，则是更新窗口的布局。窗口的布局参数发生变化时，如LayoutParams.width从100变为了200，则需要将这个变化通知给WMS使其调整Surface的大小，并让窗口进行重绘。这个工作在WindowManagerGlobal中由updateViewLayout()函数完成。

```
[WindowManagerGlobal.java-->WindowManagerGlobal.updateViewLayout()]
    publicvoid updateViewLayout(View view, ViewGroup.LayoutParams params) {
        ......// 参数检查
       final WindowManager.LayoutParams wparams =(WindowManager.LayoutParams)params;
        // 将布局参数保存到控件中
       view.setLayoutParams(wparams);
       synchronized (mLock) {
           // 获取窗口在三个数组中的索引
           int index = findViewLocked(view, true);
           ViewRootImpl root = mRoots[index];
            // 更新布局参数到数组中
           mParams[index] = wparams;
           // 调用ViewRootImpl的setLayoutParams()使得新的布局参数生效
           root.setLayoutParams(wparams, false);
        }
    }
```

更新窗口布局的工作在WindowManagerGlobal中是非常简单的，主要是保存新的布局参数，然后调用ViewRootImpl.setLayoutParams()进行更新。

6.2.3 删除窗口

接下来探讨窗口的删除操作。在了解了WindowManagerGlobal管理窗口的方式后应该可以很容易地推断出删除窗口所需要做的工作：

- 从3个数组中删除此窗口所对应的元素，包括控件、布局参数以及ViewRootImpl。
- 要求ViewRootImpl从WMS中删除对应的窗口(IWindow)，并释放一切需要回收的资源。

这个过程十分简单，这里就不引用相关的代码了。只是有一点需要说明一下：要求ViewRootImpl从WMS中删除窗口并释放资源的方法是调用ViewRootImpl.die()函数。因此可以得出这样一个结论：ViewRootImpl的生命从setView()开始，到die()结束。

6.2.4 WindowManager的总结

经过前文的分析，相信读者对WindowManager的工作原理有了深入的认识。

- 鉴于窗口布局和控件布局的一致性，WindowManager继承并实现了接口ViewManager。
- 使用者可以通过Context.getSystemService(Context.WINDOW_SERVICE)来获取一个WindowManager的实例。这个实例的真实类型是WindowManagerImpl。WindowManagerImpl一旦被创建就确定了通过它所创建的窗口所属哪块屏幕？哪个父窗口？
- WindowManagerImpl除了保存了窗口所属的屏幕以及父窗口以外，没有任何实质性的工作。窗口的管理都交由WindowManagerGlobal的实例完成。
- WindowManagerGlobal在一个进程中只有一个实例。
- WindowManagerGlobal在3个数组中统一管理整个进程中的所有窗口的信息。这些信息包括控件、布局参数以及ViewRootImpl三个元素。
- 除了管理窗口的上述3个元素以外，WindowManagerGlobal将窗口的创建、销毁与布局更新等任务交付给了ViewRootImpl完成。

说明 在实际的应用开发过程中，有时会在logcat的输出中遇到有关WindowLeaked的异常输出。WindowLeaked异常发生与WindowManagerGlobal中，其原因是Activity在destroy之前没有销毁其附属窗口，如对话框、弹出菜单等。

如此看来，WindowManager的实现仍然是很轻量的。窗口的创建、销毁与布局更新都指向了一个组件：ViewRootImpl。

6.3 深入理解ViewRootImpl

ViewRootImpl实现了ViewParent接口，作为整个控件树的根部，它是控件树正常运作的动力所在，控件的测量、布局、绘制以及输入事件的派发处理都由ViewRootImpl触发。另一方面，它是WindowManagerGlobal工作的实际实现者，因此它还需要负责与WMS交互通信以调整窗口的位置大小，以及对来自WMS的事件（如窗口尺寸改变等）作出相应的处理。

本节将对ViewRootImpl的实现做深入的探讨。

6.3.1 ViewRootImpl的创建及其重要的成员

ViewRootImpl创建于WindowManagerGlobal的addView()方法中，而调用addView()方法的线程即是此ViewRootImpl所掌控的控件树的UI线程。ViewRootImpl的构造主要是初始化了一些重要的成员，事先对这些重要的成员有个初步的认识对随后探讨ViewRootImpl的工作原理有很大的帮助。其构造函数代码如下：

```
[ViewRootImpl.java-->ViewRootImpl.ViewRootImpl()]
public ViewRootImpl(Context context, Displaydisplay) {
    /* ① 从WindowManagerGlobal中获取一个IWindowSession的实例。它是ViewRootImpl和
      WMS进行通信的代理 */
    mWindowSession= WindowManagerGlobal.getWindowSession(context.getMainLooper());
    // **②保存参数display**，在后面setView()调用中将会把窗口添加到这个Display上
    mDisplay= display;
   CompatibilityInfoHolder cih = display.getCompatibilityInfo();
   mCompatibilityInfo = cih != null ? cih : new CompatibilityInfoHolder();
    /* **③ 保存当前线程到mThread。**这个赋值操作体现了创建ViewRootImpl的线程如何成为UI主线程。
      在ViewRootImpl处理来自控件树的请求时（如请求重新布局，请求重绘，改变焦点等），会检
      查发起请求的thread与这个mThread是否相同。倘若不同则会拒绝这个请求并抛出一个异常*/
    mThread= Thread.currentThread();
    ......
    /* **④ mDirty用于收集窗口中的无效区域。**所谓无效区域是指由于数据或状态发生改变时而需要进行重绘
      的区域。举例说明，当应用程序修改了一个TextView的文字时，TextView会将自己的区域标记为无效
      区域，并通过invalidate()方法将这块区域收集到这里的mDirty中。当下次绘制时，TextView便
      可以将新的文字绘制在这块区域上 */
    mDirty =new Rect();
   mTempRect = new Rect();
    mVisRect= new Rect();
    /* **⑤ mWinFrame，描述了当前窗口的位置和尺寸。**与WMS中WindowState.mFrame保持着一致 */
   mWinFrame = new Rect();
    /* ⑥ 创建一个W类型的实例，W是IWindow.Stub的子类。即它将在WMS中作为新窗口的ID，以及接
      收来自WMS的回调*/
    mWindow= new W(this);
    ......
    /* **⑦ 创建mAttachInfo。**mAttachInfo是控件系统中很重要的对象。它存储了此当前控件树所以贴附
      的窗口的各种有用的信息，并且会派发给控件树中的每一个控件。这些控件会将这个对象保存在自己的
     mAttachInfo变量中。mAttachInfo中所保存的信息有WindowSession，窗口的实例（即mWindow），
      ViewRootImpl实例，窗口所属的Display，窗口的Surface以及窗口在屏幕上的位置等等。所以，当
      要需在一个View中查询与当前窗口相关的信息时，非常值得在mAttachInfo中搜索一下 */
   mAttachInfo = new View.AttachInfo(mWindowSession, mWindow, display,this, mHandler, thi
    /* **⑧ 创建FallbackEventHandler。**这个类如同PhoneWindowManger一样定义在android.policy
      包中，其实现为PhoneFallbackEventHandler。FallbackEventHandler是一个处理未经任何人
      消费的输入事件的场所。在6.5.4节中将会介绍它 */
   mFallbackEventHandler =PolicyManager.makeNewFallbackEventHandler(context);
    ......
    /* ⑨ 创建一个依附于当前线程，即主线程的Choreographer，用于通过VSYNC特性安排重绘行为 */
    mChoreographer= Choreographer.getInstance();
    ......
}
```

在构造函数之外，还有另外两个重要的成员被直接初始化：

- mHandler，类型为ViewRootHandler，一个依附于创建ViewRootImpl的线程，即主线程上的，用于将某些必须主线程进行的操作安排在主线程中执行。mHandler与mChoreographer的同时存在看似有些重复，其实它们拥有明确不同的分工与意义。由于mChoreographer处理消息时具有VSYNC特性，因此它主要用于处理与重绘相关的操作。但是由于mChoreographer需要等待VSYNC的垂直同步事件来触发对下一条消息的处理，因此它处理消息的及时性稍逊于mHandler。而mHandler的作用，则是为了将发生在其他线程中的事件安排在主线程上执行。所谓发生在其他线程中的事件是指来自于WMS，由继承自IWindow.Stub的mWindow引发的回调。由于mWindow是一个Binder对象的Bn端，因此这些回调发生在Binder的线程池中。而这些回调会影响到控件系统的重新测量、布局与绘制，因此需要此Handler将回调安排到主线程中。

说明 mHandler与mThread两个成员都是为了单线程模型而存在的。Android的UI操作不是线程安全的，而且很多操作也是建立在单线程的假设之上（如scheduleTraversals()）。采用单线程模型的目的是降低系统的复杂度，并且降低锁的开销。

- mSurface，类型为Surface。采用无参构造函数创建的一个Surface实例。mSurface此时是一个没有任何内容的空壳子，在 WMS通过relayoutWindow()为其分配一块Surface之前尚不能实用。

- mWinFrame、mPendingContentInset、mPendingVisibleInset以及mWidth，mHeight。这几个成员存储了窗口布局相关的信息。其中mWinFrame、mPendingConentInsets、mPendingVisibleInsets与窗口在WMS中的Frame、ContentInsets、VisibleInsets是保持同步的。这是因为这3个成员不仅会作为 relayoutWindow()的传出参数，而且ViewRootImpl在收到来自WMS的回调IWindow.Stub.resize()时，立即更新这3个成员的取值。因此这3个成员体现了窗口在WMS中的最新状态。与mWinFrame中的记录窗口在WMS中的尺寸不同的是，mWidth/mHeight记录了窗口在ViewRootImpl中的尺寸，二者在绝大多数情况下是相同的。当窗口在WMS中被重新布局而导致尺寸发生变化时，mWinFrame会首先被IWindow.Stub.resize()回调更新，此时mWinFrame便会与mWidth/mHeight产生差异。此时ViewRootImpl即可得知需要对控件树进行重新布局以适应新的窗口变化。在布局完成后，mWidth/mHeight会被赋值为mWinFrame中所保存的宽和高，二者重新统一。在随后分析performTraversals()方法时，读者将会看到这一处理。另外，与mWidth/mHeight类似，ViewRootImpl也保存了窗口的位置信息Left/Top以及ContentInsets/VisibleInsets供控件树查询，不过这四项信息被保存在了mAttachInfo中。

ViewRootImpl的在其构造函数中初始化了一系列的成员变量，然而其创建过程仍未完成。仅在为其指定了一个控件树进行管理，并向WMS添加了一个新的窗口之后，ViewRootImpl承上启下的角色才算完全确立下来。因此需要进一步分析ViewRootImpl.setView()方法。

```
[ViewRootImp.java-->ViewRootImpl.setView()]
public void setView(View view,WindowManager.LayoutParams attrs, View panelParentView) {
   synchronized (this) {
        if (mView == null) {
           // **① mView保存了控件树的根**
           mView = view;
           ......
           // ②mWindowAttributes保存了窗口所对应的LayoutParams
           mWindowAttributes.copyFrom(attrs);
            ......
           /* 在添加窗口之前，先通过requestLayout()方法在主线程上安排一次“遍历”。所谓
             “遍历”是指ViewRootImpl中的核心方法performTraversals()。这个方法实现了对
              控件树进行测量、布局、向WMS申请修改窗口属性以及重绘的所有工作。由于此“遍历”
              操作对于初次遍历做了一些特殊处理，而来自WMS通过mWindow发生的回调会导致一些属性
              发生变化，如窗口的尺寸、Insets以及窗口焦点等，从而有可能使得初次“遍历”的现场遭
              到破坏。因此，需要在添加窗口之前，先发送一个“遍历”消息到主线程。
               在主线程中向主线程的Handler发送消息如果使用得当，可以产生很精妙的效果。例如本例
              中可以实现如下的执行顺序：添加窗口->初次遍历->处理来自WMS的回调 */
           requestLayout();
           /***③ 初始化mInputChannel。**参考第五章，InputChannel是窗口接受来自InputDispatcher
             的输入事件的管道。 注意，仅当窗口的属性inputFeatures不含有
             INPUT_FEATURE_NO_INPUT_CHANNEL时才会创建InputChannel，否则mInputChannel
             为空，从而导致此窗口无法接受任何输入事件 */
           if ((mWindowAttributes.inputFeatures
                   & WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANNEL) == 0) {
               mInputChannel = new InputChannel();
           }
           try {
               ......
               /* 将窗口添加到WMS中。完成这个操作之后，mWindow已经被添加到指定的Display中去
                 而且mInputChannel（如果不为空）已经准备好接受事件了。只是由于这个窗口没有进行
                 过relayout()，因此它还没有有效的Surface可以进行绘制 */
               res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
                        getHostVisibility(), mDisplay.getDisplayId(),
                       mAttachInfo.mContentInsets, mInputChannel);
           } catch (RemoteException e) {......} finally { ...... }
           ......
           if (res &lt; WindowManagerGlobal.ADD_OKAY) {
               // 错误处理。窗口添加失败的原因通常是权限问题，重复添加，或者tokeen无效
           }
           ......
            /*④ 如果mInputChannel不为空，则创建mInputEventReceiver，用于接受输入事件。
             注意第二个参数传递的是Looper.myLooper()，即mInputEventReceiver将在主线程上
             触发输入事件的读取与onInputEvent()。这是应用程序可以在onTouch()等事件响应中
             直接进行UI操作等根本原因。
            */
           if (mInputChannel != null) {
               ......
               mInputEventReceiver = new WindowInputEventReceiver(mInputChannel,
                           Looper.myLooper());
           }
          /* ViewRootImpl将作为参数view的parent。所以，ViewRootImpl可以从控件树中任何一个
            控件开始，通过回溯getParent()的方法得到 */
           view.assignParent(this);
            ......
        }
    }
}
```

至此，ViewRootImpl所有重要的成员都已经初始化完毕，新的窗口也已经添加到WMS中。ViewRootImpl的创建过程是由构造函数和setView()方法两个环节构成的。其中构造函数主要进行成员的初始化，setView()则是创建窗口、建立输入事件接收机制的场所。同时，触发第

一次“遍历”操作的消息已经发送给主线程，在随后的第一次“遍历”完成后，ViewRootImpl将会完成对控件树的第一次测量、布局，并从WMS获取窗口的Surface以进行控件树的初次绘制工作。

在本节的最后，通过图 6 – 4对ViewRootImpl中的重要成员进行了分类整理。



图 6 - 4 ViewRootImpl中的主要成员

6.3.2 控件系统的心跳：performTraversals()

ViewRootImpl在其创建过程中通过requestLayout()向主线程发送了一条触发“遍历”操作的消息，“遍历”操作是指performTraversals()方法。它的性质与WMS中的performLayoutAndPlaceSurfacesLocked()类似，是一个包罗万象的方法。ViewRootImpl中接收到的各种变化，如来自WMS的窗口属性变化，来自控件树的尺寸变化、重绘请求等都引发performTraversals()的调用，并在其中完成处理。View类及其子类中的onMeasure()、onLayout()以及onDraw()等回调也都是在performTraversals()的执行过程中直接或间接地引发。也正是如此，一次次的performTraversals()调用驱动着控件树有条不紊地工作着，一旦此方法无法正常执行，整个控件树都将处于僵死状态。因此，performTraversals()函数可谓是ViewRootImpl的心跳。

由于布局的相关工作是此方法中最主要的内容，为了简化分析，并突出此方法的工作流程，本节将以布局的相关工作为主线进行探讨。待完成了这部分内容的分析之后，庞大的performTraversals()方法将不再那么难以驯服，读者便可以轻易地学习其他的工作了。

1. performTraversals()的工作阶段

performTraversals()是Android 源码中最庞大的方法之一，因此在正式探讨它的实现之前最好先将其划分为以下几个工作阶段作为指导。

- 预测量阶段。这是进入performTraversals()方法后的第一个阶段，它会对控件树进行第一次测量。测量结果可以通过mView. getMeasuredWidth()/Height()获得。在此阶段中将会计算出控件树为显示其内容所需的尺寸，即期望的窗口尺寸。在这个阶段中，View及其子类的onMeasure()方法将会沿着控件树依次得到回调。

- 布局窗口阶段。根据预测量的结果，通过IWindowSession.relayout()方法向WMS请求调整窗口的尺寸等属性，这将引发WMS对窗口进行重新布局，并将布局结果返回给ViewRootImpl。

- 最终测量阶段。预测量的结果是控件树所期望的窗口尺寸。然而由于在WMS中影响窗口布局的因素很多(参考第4章)，WMS不一定会将窗口准确地布局为控件树所要求的尺寸，而迫于WMS作为系统服务的强势地位，控件树不得不接受WMS的布局结果。因此在这一阶段，performTraversals()将以窗口的实际尺寸对控件进行最终测量。在这个阶段中，View及其子类的onMeasure()方法将会沿着控件树依次被回调。

- 布局控件树阶段。完成最终测量之后便可以对控件树进行布局了。测量确定的是控件的尺寸，而布局则是确定控件的位置。在这个阶段中，View及其子类的onLayout()方法将会被回调。

- 绘制阶段。这是performTraversals()的最终阶段。确定了控件的位置与尺寸后，便可以对控件树进行绘制了。在这个阶段中，View及其子类的onDraw()方法将会被回调。

说明 很多文章都倾向于将performTraversals()的工作划分为测量、布局与绘制三个阶段。然而笔者认为如此划分隐藏了WMS在这个过程中的地位，并且没能体现出控件树对窗口尺寸的期望、WMS对窗口尺寸做最终的确定，最后以WMS给出的结果为准再次进行测量的协商过程。而这个协商过程充分体现了ViewRootImpl作为WMS与控件树的中间人的角色。

接下来将结合代码，对上述五个阶段进行深入的分析。

2. 预测量与测量原理

本节将探讨performTraversals()将以何种方式对控件树进行预测量，同时，本节也会对控件的测量过程与原理进行介绍。

预测量参数的候选

预测量也是一次完整的测量过程，它与最终测量的区别仅在于参数不同而已。实际的测量工作在View或其子类的onMeasure()方法中完成，并且其测量结果需要受限于来自其父控件的指示。这个指示由onMeasure()方法的两个参数进行传达：widthSpec与heightSpec。它们是被称为MeasureSpec的复合整型变量，用于指导控件对自身进行测量。它有两个分量，结构如图6-5所示。



图 6 - 5 MeasureSpec的结构

其1到30位给出了父控件建议尺寸。建议尺寸对测量结果的影响依不同的SPEC_MODE的不同而不同。SPEC_MODE的取值取决于此控件的LayoutParams.width/height的设置，可以是如下三种值之一。

- MeasureSpec.UNSPECIFIED (0)：表示控件在进行测量时，可以无视SPEC_SIZE的值。控件可以是它所期望的任意尺寸。
- MeasureSpec.EXACTLY (1)：表示子控件必须为SPEC_SIZE所制定的尺寸。当控件的LayoutParams.width/height为一确定值，或者是MATCH_PARENT时，对应的MeasureSpec参数会使用这个SPEC_MODE。
- MeasureSpec.AT_MOST (2)：表示子控件可以是它所期望的尺寸，但是不得大于SPEC_SIZE。当控件的LayoutParams.width/height为WRAP_CONTENT时，对应的MeasureSpec参数会使用这个SPEC_MODE。

Android提供了一个MeasureSpec类用于组合两个分量成为一个MeasureSpec，或者从MeasureSpec中分离任何一个分量。

那么ViewRootImpl会如何为控件树的根mView准备其MeasureSpec呢？

参考如下代码，注意desiredWindowWidth/Height的取值，它们将是SPEC_SIZE分量的候选。另外，这段代码分析中也解释了与测量无关，但是比较重要的代码段。

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()]
private void performTraversals() {
    // 将mView保存在局部变量host中，以此提高对mView的访问效率
    finalView host = mView;
    ......
    // 声明本阶段的主角，这两个变量将是mView的SPEC_SIZE分量的候选
    intdesiredWindowWidth;
    intdesiredWindowHeight;
    .......
    Rectframe = mWinFrame; // 如上一节所述，mWinFrame表示了窗口的最新尺寸
    if(mFirst) {
        /*mFirst表示了这是第一次遍历，此时窗口刚刚被添加到WMS，此时窗口尚未进行relayout，因此
          mWinFrame中没有存储有效地窗口尺寸 */
        if(lp.type == WindowManager.LayoutParams.TYPE_STATUS_BAR_PANEL) {
            ......// 为状态栏设置desiredWindowWidth/Height，其取值是屏幕尺寸
        }else {
            //① 第一次“遍历”的测量，采用了应用可以使用的最大尺寸作为SPEC_SIZE的候选
           DisplayMetrics packageMetrics =
               mView.getContext().getResources().getDisplayMetrics();
           desiredWindowWidth = packageMetrics.widthPixels;
           desiredWindowHeight = packageMetrics.heightPixels;
        }
        /* 由于这是第一次进行“遍历”，控件树即将第一次被显示在窗口上，因此接下来的代码填充了
```

```
        mAttachInfo中的一些字段，然后通过mView发起了dispatchAttachedToWindow()的调用
         之后每一个位于控件树中的控件都会回调onAttachedToWindow() */
        ......
    } else {
        // ② 在非第一次遍历的情况下，会采用窗口的最新尺寸作为SPEC_SIZE的候选
       desiredWindowWidth = frame.width();
       desiredWindowHeight = frame.height();
        /* 如果窗口的最新尺寸与ViewRootImpl中的现有尺寸不同，说明WMS侧单方面改变了窗口的尺寸
          这将产生如下三个结果 */
        if(desiredWindowWidth != mWidth || desiredWindowHeight != mHeight) {
            // 需要进行完整的重绘以适应新的窗口尺寸
           mFullRedrawNeeded = true;
           // 需要对控件树进行重新布局
           mLayoutRequested = true;
           /* 控件树有可能拒绝接受新的窗口尺寸，比如在随后的预测量中给出了不同于窗口尺寸的测量结果
             产生这种情况时，就需要在窗口布局阶段尝试设置新的窗口尺寸 */
           windowSizeMayChange = true;
        }
    }
    ......
    /* 执行位于RunQueue中的回调。RunQueue是ViewRootImpl的一个静态成员，即是说它是进程唯一
      的，并且可以在进程的任何位置访问RunQueue。在进行多线程任务时，开发者可以通过调用View.post()
      或View.postDelayed()方法将一个Runnable对象发送到主线程执行。这两个方法的原理是将
      Runnable对象发送到ViewRootImpl的mHandler去。当控件已经加入到控件树时，可以通过
     AttachInfo轻易获取这个Handler。而当控件没有位于控件树中时，则没有mAttachInfo可用，此时
      执行View.post()/PostDelay()方法，Runnable将会被添加到这个RunQueue队列中。
      在这里，ViewRootImpl将会把RunQueue中的Runnable发送到mHandler中，进而得到执行。所以
      无论控件是否显示在控件树中，View.post()/postDelay()方法都是可用的，除非当前进程中没有任何
      处于活动状态的ViewRootImpl */
   getRunQueue().executeActions(attachInfo.mHandler);
    booleanlayoutRequested = mLayoutRequested && !mStopped;
    /* 仅当layoutRequested为true时才进行预测量。
     layoutRequested为true表示在进行“遍历”之前requestLayout()方法被调用过。
     requestLayout()方法用于要求ViewRootImpl进行一次“遍历”并对控件树重新进行测量与布局 */
    if(layoutRequested) {
       final Resources res = mView.getContext().getResources();
        if(mFirst) {
            ......// 确定控件树是否需要进入TouchMode，本章将在6.5.1节介绍 TouchMode
        }else {
            /*检查WMS是否单方面改变了ContentInsets与VisibleInsets。注意对二者的处理的差异，
             ContentInsets描述了控件在布局时必须预留的空间，这样会影响控件树的布局，因此将
             insetsChanged标记为true，以此作为是否进行控件布局的条件之一。而VisibleInsets则
             描述了被遮挡的空间，ViewRootImpl在进行绘制时，需要调整绘制位置以保证关键控件或区域，
             如正在进行输入的TextView等不被遮挡，这样VisibleInsets的变化并不会导致重新布局，
             所以这里仅仅是将VisibleInsets保存到mAttachInfo中，以便绘制时使用 */
           if (!mPendingContentInsets.equals(mAttachInfo.mContentInsets)) {
               insetsChanged = true;
           }
           if (!mPendingVisibleInsets.equals(mAttachInfo.mVisibleInsets)) {
               mAttachInfo.mVisibleInsets.set(mPendingVisibleInsets);
           }
            /*当窗口的width或height被指定为WRAP_CONTENT时，表示这是一个悬浮窗口。
             此时会对desiredWindowWidth/Height进行调整。在前面的代码中，这两个值被设置
             被设置为窗口的当前尺寸。而根据MeasureSpec的要求，测量结果不得大于SPEC_SIZE。
             然而，如果这个悬浮窗口需要更大的尺寸以完整显示其内容时，例如为AlertDialog设置了
             一个更长的消息内容，如此取值将导致无法得到足够大的测量结果，从而导致内容无法完整显示。
             因此，对于此等类型的窗口，ViewRootImpl会调整desiredWindowWidth/Height为此应用
             可以使用的最大尺寸 */
           if (lp.width == ViewGroup.LayoutParams.WRAP_CONTENT
                   || lp.height == ViewGroup.LayoutParams.WRAP_CONTENT) {
               // 悬浮窗口的尺寸取决于测量结果。因此有可能需要向WMS申请改变窗口的尺寸。
               windowSizeMayChange = true;
               if (lp.type == WindowManager.LayoutParams.TYPE_STATUS_BAR_PANEL) {
                   //
               } else {
                   // ③ 设置悬浮窗口SPEC_SIZE的候选为应用可以使用的最大尺寸
                   DisplayMetrics packageMetrics = res.getDisplayMetrics();
                   desiredWindowWidth = packageMetrics.widthPixels;
                   desiredWindowHeight = packageMetrics.heightPixels;
               }
           }
        }
```

```
            // **④ 进行预测量。**通过measureHierarchy()方法以desiredWindowWidth/Height进行测量
            windowSizeMayChange |=measureHierarchy(host, lp, res,
                    desiredWindowWidth, desiredWindowHeight);
        }
        // 其他阶段的处理
        ......
}
```

由此可知，预测量时的SPEC_SIZE按照如下原则进行取值：

- 第一次"遍历"时，使用应用可用的最大尺寸作为SPEC_SIZE的候选。
- 此窗口是一个悬浮窗口，即LayoutParams.width/height其中之一被指定为WRAP_CONTENT时，使用应用可用的最大尺寸作为SPEC_SIZE的候选。
- 在其他情况下，使用窗口最新尺寸作为SPEC_SIZE的候选。

最后，通过measureHierarchy()方法进行测量。

测量协商

measureHierarchy()用于测量整个控件树。传入的参数desiredWindowWidth与desiredWindowHeight在前述代码中根据不同的情况作了精心的挑选。控件树本可以按照这两个参数完成测量，但是measureHierarchy()有自己的考量，即如何将窗口布局地尽可能地优雅。

这是针对将LayoutParams.width设置为了WRAP_CONTENT的悬浮窗口而言。如前文所述，在设置为WRAP_CONTENT时，指定的desiredWindowWidth是应用可用的最大宽度，如此可能会产生如图6-6左图所示的丑陋布局。这种情况较容易发生在AlertDialog中，当AlertDialog需要显示一条比较长的消息时，由于给予的宽度足够大，因此它有可能将这条消息以一行显示，并使得其窗口充满了整个屏幕宽度，在横屏模式下这种布局尤为丑陋。

倘若能够对可用宽度进行适当的限制，迫使AlertDialog将消息换行显示，则产生的布局结果将会优雅得多，如图6-6右图所示。但是，倘若不分清红皂白地对宽度进行限制，当控件树真正需要足够的横向空间时，会导致内容无法显示完全，或者无法达到最佳的显示效果。例如当一个悬浮窗口希望尽可能大地显示一张照片时就会出现这样的情况。



图 6 - 6 丑陋的布局与优雅的布局

那么measureHierarchy()如何解决这个问呢？它采取了与控件树进行协商的办法，即先使用measureHierarchy()所期望的宽度限制尝试对控件树进行测量，然后通过测量结果来检查控件树是否能够在此限制下满足其充分显示内容的要求。倘若没能满足，则measureHierarchy()进行让步，放宽对宽度的限制，然后再次进行测量，再做检查。倘若仍不能满足则再度进行让步。

参考代码如下：

```
[ViewRootImpl.java-->ViewRootImpl.measureHierarchy()]
private boolean measureHierarchy(final View host,final WindowManager.LayoutParams lp,
       final Resources res, final int desiredWindowWidth,
        final int desiredWindowHeight) {
    intchildWidthMeasureSpec; // 合成后的用于描述宽度的MeasureSpec
    intchildHeightMeasureSpec; // 合成后的用于描述高度的MeasureSpec
    booleanwindowSizeMayChange = false; // 表示测量结果是否可能导致窗口的尺寸发生变化
    booleangoodMeasure = false; // goodMeasure表示了测量是否能满足控件树充分显示内容的要求
    // 测量协商仅发生在LayoutParams.width被指定为WRAP_CONTENT的情况下
    if(lp.width == ViewGroup.LayoutParams.WRAP_CONTENT) {
        /* **① 第一次协商。**measureHierarchy()使用它最期望的宽度限制进行测量。这一宽度限制定义为
         一个系统资源。可以在frameworks/base/core/res/res/values/config.xml找到它的定义 */
       res.getValue(com.android.internal.R.dimen.config_prefDialogWidth,mTmpValue, true);
        intbaseSize = 0;
        // 宽度限制被存放在baseSize中
        if(mTmpValue.type == TypedValue.TYPE_DIMENSION) {
           baseSize = (int)mTmpValue.getDimension(packageMetrics);
        }
        if(baseSize != 0 && desiredWindowWidth &gt; baseSize) {
           // 使用getRootMeasureSpec()函数组合SPEC_MODE与SPEC_SIZE为一个MeasureSpec
           childWidthMeasureSpec = getRootMeasureSpec(baseSize, lp.width);
            childHeightMeasureSpec =
                           getRootMeasureSpec(desiredWindowHeight,lp.height);
            //**②第一次测量。**由performMeasure()方法完成
           performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
           /* 控件树的测量结果可以通过mView的getmeasuredWidthAndState()方法获取。如果
             控件树对这个测量结果不满意，则会在返回值中添加MEASURED_STATE_TOO_SMALL位 */
           if ((host.getMeasuredWidthAndState()&View.MEASURED_STATE_TOO_SMALL)
                                                     ==0) {
               goodMeasure = true; // 控件树对测量结果满意，测量完成
           } else {
               // **③ 第二次协商。**上次测量结果表明控件树认为measureHierarchy()给予的宽度太小，
                 在此适当地放宽对宽度的限制，使用最大宽度与期望宽度的中间值作为宽度限制 */
               baseSize = (baseSize+desiredWindowWidth)/2;
               childWidthMeasureSpec = getRootMeasureSpec(baseSize, lp.width);
               // **④ 第二次测量**
               performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
               // 再次检查控件树是否满足此次测量
               if ((host.getMeasuredWidthAndState()&View.MEASURED_STATE_TOO_SMALL)
                                                                           == 0) {
                   goodMeasure = true; // 控件树对测量结果满意，测量完成
               }
            }
        }
    }
    if(!goodMeasure) {
        /* **⑤ 最终测量。**当控件树对上述两次协商的结果都不满意时，measureHierarchy()放弃所有限制
          做最终测量。这一次将不再检查控件树是否满意了，因为即便其不满意，measurehierarchy()也没
          有更多的空间供其使用了 */
       childWidthMeasureSpec = getRootMeasureSpec(desiredWindowWidth,lp.width);
       childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight,lp.height);
       performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
        /* 最后，如果测量结果与ViewRootImpl中当前的窗口尺寸不一致，则表明随后可能有必要进行窗口
          尺寸的调整 */
        if(mWidth != host.getMeasuredWidth() || mHeight != host.getMeasuredHeight())
        {
           windowSizeMayChange = true;
        }
    }
    // 返回窗口尺寸是否可能需要发生变化
    returnwindowSizeMayChange;
}
```

显然，对于非悬浮窗口，即当LayoutParams.width被设置为MATCH_PARENT时，不存在协商过程，直接使用给定的desiredWindowWidth/Height进行测量即可。而对于悬浮窗口，measureHierarchy()可以连续进行两次让步。因而在最不利的情况下，在ViewRootImpl的一

次"遍历"中，控件树需要进行三次测量，即控件树中的每一个View.onMeasure()会被连续调用三次之多，如图6-7所示。所以相对于onLayout()，onMeasure()方法的对性能的影响比较大。



图 6 - 7 协商测量的三次尝试

接下来通过performMeasure()看控件树如何进行测量。

测量原理

performMeasure()方法的实现非常简单，它直接调用mView.measure()方法，将measureHierarchy()给予的widthSpec与heightSpec交给mView。

看下View.measure()方法的实现：

```
[View.java-->View.measure()]
public final void measure(int widthMeasureSpec,int heightMeasureSpec) {
    /* 仅当给予的MeasureSpec发生变化，或要求强制重新布局时，才会进行测量。
      所谓强制重新布局，是指当控件树中的一个子控件的内容发生变化时，需要进行重新的测量和布局的情况
      在这种情况下，这个子控件的父控件（以及其父控件的父控件）所提供的MeasureSpec必定与上次测量
      时的值相同，因而导致从ViewRootImpl到这个控件的路径上的父控件的measure()方法无法得到执行
      进而导致子控件无法重新测量其尺寸或布局。因此，当子控件因内容发生变化时，从子控件沿着控件树回溯
      到ViewRootImpl，并依次调用沿途父控件的requestLayout()方法，在这个方法中，会在
     mPrivateFlags中加入标记PFLAG_FORCE_LAYOUT，从而使得这些父控件的measure()方法得以顺利
      执行，进而这个子控件有机会进行重新测量与布局。这便是强制重新布局的意义 */
    if ((mPrivateFlags& PFLAG_FORCE_LAYOUT) == PFLAG_FORCE_LAYOUT ||
           widthMeasureSpec != mOldWidthMeasureSpec ||
           heightMeasureSpec != mOldHeightMeasureSpec) {
        /* **① 准备工作。**从mPrivateFlags中将PFLAG_MEASURED_DIMENSION_SET标记去除。
          PFLAG_MEASURED_DIMENSION_SET标记用于检查控件在onMeasure()方法中是否通过
          调用setMeasuredDimension()将测量结果存储下来 */
       mPrivateFlags &= ~PFLAG_MEASURED_DIMENSION_SET;
       ......
        /* **② 对本控件进行测量** 每个View子类都需要重载这个方法以便正确地对自身进行测量。
           View类的onMeasure()方法仅仅根据背景Drawable或style中设置的最小尺寸作为
           测量结果*/
       onMeasure(widthMeasureSpec, heightMeasureSpec);
        /* ③ 检查onMeasure()的实现是否调用了setMeasuredDimension()
          setMeasuredDimension()会将PFLAG_MEASURED_DIMENSION_SET标记重新加入
          mPrivateFlags中。之所以做这样的检查，是由于onMeasure()的实现可能由开发者完成，
           而在Android看来，开发者是不可信的 */
        if((mPrivateFlags & PFLAG_MEASURED_DIMENSION_SET)
                                   !=PFLAG_MEASURED_DIMENSION_SET) {
           throw new IllegalStateException(......);
        }
       // ④ 将PFLAG_LAYOUT_REQUIRED标记加入mPrivateFlags。这一操作会对随后的布局操作放行
       mPrivateFlags |= PFLAG_LAYOUT_REQUIRED;
    }
    // 记录父控件给予的MeasureSpec，用以检查之后的测量操作是否有必要进行
   mOldWidthMeasureSpec = widthMeasureSpec;
   mOldHeightMeasureSpec = heightMeasureSpec;
}
```

从这段代码可以看出，View.measure()方法没有实现任何测量算法，它的作用在于引发onMeasure()的调用，并对onMeasure()行为的正确性进行检查。另外，在控件系统看来，一旦控件执行了测量操作，那么随后必须进行布局操作，因此在完成测量之后，将PFLAG_LAYOUT_REQUIRED标记加入mPrivateFlags，以便View.layout()方法可以顺利进行。

onMeasure()的结果通过setMeasuredDimension()方法尽行保存。setMeasuredDimension()方法的实现如下：

```
[View.java-->View.setMeasuredDimension()]
protected final void setMeasuredDimension(intmeasuredWidth, int measuredHeight) {
    /* ① 测量结果被分别保存在成员变量mMeasuredWidth与mMeasuredHeight中
   mMeasuredWidth = measuredWidth;
    mMeasuredHeight = measuredHeight;
    // ② 向mPrivateFlags中添加PFALG_MEASURED_DIMENSION_SET，以此证明onMeasure()保存了测量结果
   mPrivateFlags |= PFLAG_MEASURED_DIMENSION_SET;
}
```

其实现再简单不过。存储测量结果的两个变量可以通过getMeasuredWidthAndState()与getMeasuredHeightAndState()两个方法获得，就像ViewRootImpl.measureHierarchy()中所做的一样。此方法虽然简单，但需要注意，与MeasureSpec类似，测量结果不仅仅是一个尺寸，而是一个测量状态与尺寸的复合整、变量。其0至30位表示了测量结果的尺寸，而31、32位则表示了控件对测量结果是否满意，即父控件给予的MeasureSpec是否可以使得控件完整地显示其内容。当控件对测量结果满意时，直接将尺寸传递给setMeasuredDimension()即可，注意要保证31、32位为0。倘若对测量结果不满意，则使用View.MEASURED_STATE_TOO_SMALL | measuredSize 作为参数传递给setMeasuredDimension()以告知父控件对MeasureSpec进行可能的调整。

既然明白了onMeasure()的调用如何发起，以及它如何将测量结果告知父控件，那么onMeasure()方法应当如何实现的呢？对于非ViewGroup的控件来说其实现相对简单，只要按照MeasureSpec的原则如实计算其所需的尺寸即可。而对于ViewGroup类型的控件来说情况则复杂得多，因为它不仅拥有自身需要显示的内容（如背景），它的子控件也是其需要测量的内容。因此它不仅需要计算自身显示内容所需的尺寸，还有考虑其一系列子控件的测量结果。为此它必须为每一个子控件准备MeasureSpec，并调用每一个子控件的measure()函数。

由于各种控件所实现的效果形形色色，开发者还可以根据需求自行开发新的控件，因此onMeasure()中的测量算法也会变化万千。不从Android系统实现的角度仍能得到如下的onMeasure()算法的一些实现原则：

- 控件在进行测量时，控件需要将它的Padding尺寸计算在内，因为Padding是其尺寸的一部分。

- ViewGroup在进行测量时，需要将子控件的Margin尺寸计算在内。因为子控件的Margin尺寸是父控件尺寸的一部分。

- ViewGroup为子控件准备MeasureSpec时，SPEC_MODE应取决于子控件的LayoutParams.width/height的取值。取值为MATCH_PARENT或一个确定的尺寸时应为EXACTLY，WRAP_CONTENT时应为AT_MOST。至于SPEC_SIZE，应理解为ViewGroup对子控件尺寸的限制，即ViewGroup按照其实现意图所允许子控件获得的最大尺寸。并且需要扣除子控件的Margin尺寸。
- 虽然说测量的目的在于确定尺寸，与位置无关。但是子控件的位置是ViewGroup进行测量时必须要首先考虑的。因为子控件的位置即决定了子控件可用的剩余尺寸，也决定了父控件的尺寸（当父控件的LayoutParams.width/height为WRAP_CONTENT时）。
- 在测量结果中添加MEASURED_STATE_TOO_SMALL需要做到实事求是。当一个方向上的空间不足以显示其内容时应考虑利用另一个方向上的空间，例如对文字进行换行处理，因为添加这个标记有可能导致父控件对其进行重新测量从而降低效率。
- 当子控件的测量结果中包含MEASURED_STATE_TOO_SMALL标记时，只要有可能，父控件就应当调整给予子控件的MeasureSpec，并进行重新测量。倘若没有调整的余地，父控件也应当将MEASURED_STATE_TOO_SMALL加入到自己的测量结果中，让它的父控件尝试进行调整。
- ViewGroup在测量子控件时必须调用子控件的measure()方法，而不能直接调用其onMeasure()方法。直接调用onMeasure()方法的最严重后果是子控件的PFLAG_LAYOUT_REQUIRED标识无法加入到mPrivateFlag中，从而导致子控件无法进行布局。

综上所述，测量控件树的实质是测量控件树的根控件。完成控件树的测量之后，ViewRootImpl便得知了控件树对窗口尺寸的需求。

确定是否需要改变窗口尺寸

接下来回到performTraversals()方法。在ViewRootImpl.measureHierarchy()执行完毕之后，ViewRootImpl了解了控件树所需的空间。于是便可确定是否需要改变窗口窗口尺寸以便满足控件树的空间要求。前述的代码中多处设置windowSizeMayChange变量为true。windowSizeMayChange仅表示有可能需要改变窗口尺寸。而接下来的这段代码则用来确定窗口是否需要改变尺寸。

```
[ViewRootImpl.java-->ViewRootImp.performTraversals()]
private void performTraversals() {
    ......// 测量控件树的代码
    /* 标记mLayoutRequested为false。因此在此之后的代码中，倘若控件树中任何一个控件执行了
     requestLayout()，都会重新进行一次“遍历” */
    if (layoutRequested) {
        mLayoutRequested = false;
    }
    // 确定窗口是否确实需要进行尺寸的改变
    booleanwindowShouldResize = layoutRequested && windowSizeMayChange
       && ((mWidth != host.getMeasuredWidth() || mHeight !=host.getMeasuredHeight())
           || (lp.width == ViewGroup.LayoutParams.WRAP_CONTENT &&
                   frame.width() &lt; desiredWindowWidth && frame.width() !=mWidth)
           || (lp.height == ViewGroup.LayoutParams.WRAP_CONTENT &&
                   frame.height() &lt; desiredWindowHeight && frame.height() !=mHeight));
}
```

确定窗口尺寸是否确实需要改变的条件看起来比较复杂，这里进行一下总结，先介绍必要条件：

- layoutRequested为true，即ViewRootImpl.requestLayout()方法被调用过。View中也有requestLayout()方法。当控件内容发生变化从而需要调整其尺寸时，会调用其自身的requestLayout()，并且此方法会沿着控件树向根部回溯，最终调用到ViewRootImp.requestLayout()，从而引发一次performTraversals()调用。之所以这是一个必要条件，是因为performTraversals()还有可能因为控件需要重绘时被调用。当控件仅需要重绘而不需要重新布局时（例如背景色或前景色发生变化时），会通过invalidate()方法回溯到ViewRootImpl，此时不会通过performTraversals()触发performTraversals()调用，而是通过scheduleTraversals()进行触发。在这种情况下layoutRequested为false，即表示窗口尺寸不需发生变化。

- windowSizeMayChange为true，如前文所讨论的，这意味着WMS单方面改变了窗口尺寸而控件树的测量结果与这一尺寸有差异，或当前窗口为悬浮窗口，其控件树的测量结果将决定窗口的新尺寸。

在满足上述两个条件的情况下，以下两个条件满足其一：

- 测量结果与ViewRootImpl中所保存的当前尺寸有差异。

- 悬浮窗口的测量结果与窗口的最新尺寸有差异。

注意ViewRootImpl对是否需要调整窗口尺寸的判断是非常小心的。第4章介绍WMS的布局子系统时曾经介绍过，调整窗口尺寸所必须调用的performLayoutAndPlaceSurfacesLocked()函数会导致WMS对系统中的所有窗口新型重新布局，而且会引发至少一个动画帧渲染，其计算开销相当之大。因此ViewRootImpl仅在必要时才会惊动WMS。

至此，预测量阶段完成了。

总结

这一阶段的工作内容是为了给后续阶段做参数的准备并且其中最重要的工作是对控件树的预测量，至此ViewRootImpl得知了控件树对窗口尺寸的要求。另外，这一阶段还准备了后续阶段所需的其他参数：

- viewVisibilityChanged。即View的可见性是否发生了变化。由于mView是窗口的内容，因此mView的可见性即是窗口的可见性。当这一属性发生变化时，需要通过通过WMS改变窗口的可见性。

LayoutParams。预测量阶段需要收集应用到LayoutParams的改动，这些改动一方面来自于WindowManager.updateViewLayout()，而另一方面则来自于控件树。以SystemUIVisibility为例，View.setSystemUIVisibility()所修改的设置需要反映到LayoutParams中，而这些设置确却保存在控件自己的成员变量里。在预测量阶段会通过ViewRootImpl.collectViewAttributes()方法遍历控件树中的所有控件以收集这些设置，然后更新LayoutParams。

第7章 深入理解SystemUI（节选）

本章主要内容：

- 探讨状态栏与导航栏的启动过程
- 介绍状态栏中的通知信息、系统状态图标等信息的管理与显示原理
- 介绍导航栏中的虚拟按键、SearchPanel的工作原理
- 介绍SystemUIVisibility

本章涉及的源代码文件名及位置：

- SystemServer.java

frameworks/base/services/java/com/android/server/SystemServer.java

- SystemUIService.java

frameworks/base/packages/SystemUI/src/com/android/systemui/SystemUIService.java

- PhoneWindowManager.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java

- PhoneStatusBar.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/PhoneStatusBar.java

- BaseStatusBar.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/BaseStatusBar.java

- StatusBarManager.java

frameworks/base/core/java/android/app/StatusBarManager.java

- StatusBarManagerService.java

frameworks/base/services/java/com/android/server/StatusBarManagerService.java

- NotificationManager.java

frameworks/base/core/java/android/app/NotificationManager.java

- NotificationManagerService.java

frameworks/base/services/java/com/android/server/NotificationManagerService.java

- KeyButtonView.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/policy/KeyButtonView.java

- NavigationBarView.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/NavigationBarView.java

- DelegateViewHelper.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/DelegateViewHelper.java

- SearchPanelView.java

frameworks/base/packages/SystemUI/src/com/android/systemui/SearchPanelView.java

- PhoneWindow.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java

- InputMethodService.java

frameworks/base/core/java/android/inputmethodservice/InputMethodService.java

- View.java

frameworks/base/core/java/android/view/View.java

- ViewRootImpl.java

frameworks/base/core/java/android/view/ViewRootImpl.java

- WindowManagerService.java

frameworks/base/services/java/com/android/server/wm/WindowManagerService.java

7.1初识SystemUI

顾名思义，SystemUI是为用户提供系统级别的信息显示与交互的一套UI组件，因此它所实现的功能包罗万象。屏幕顶端的状态栏、底部的导航栏、图片壁纸以及RecentPanel（近期使用的APP列表）都属于SystemUI的范畴。SystemUI中还有一个名为TakeScreenshotService的服务，用于在用户按下音量下键与电源键时进行截屏操作。在第5章曾介绍了PhoneWindowManager监听这一组合键的机制，当它捕捉到这一组合键时便会向

TakeScreenShotService发送请求从而完成截屏操作。SystemUI还提供了PowerUI和RingtonePlayer两个服务。前者负责监控系统的剩余电量并在必要时为用户显示低电警告，后者则依托AudioService为向其他应用程序提供播放铃声的功能。SystemUI的博大不止如此，读者可以通过查看其AndroidManifest.xml来了解它所实现的其他功能。本章将着重介绍其中最重要的两个功能的实现：状态栏和导航栏。

7.1.1 SystemUIService的启动

尽管SystemUI的表现形式与普通的Android应用程序大相径庭，但它却是以一个APK的形式存在于系统之中，即它与普通的Android应用程序并没有本质上的区别。无非是通过Android四大组件中的Activity、Service、BroadcastReceiver接受外界的请求并执行相关的操作，只不过它们所接受到的请求主要来自各个系统服务而已。

SystemUI包罗万象，并且大部分功能之间相互独立，比如RecentPanel、TakeScreenshotService等均是按需启动，并在完成其既定任务后退出，这与普通的Activity以及Service别无二致。比较特殊的是状态栏、导航栏等组件的启动方式。它们运行于一个称之为SystemUIService的一个Service之中。因此讨论状态栏与导航栏的启动过程其实就是SystemUIService的启动过程。

1. SystemUIService的启动时机

那么SystemUIService在何时由谁启动的呢？作为一个系统级别的UI组件，自然要在系统的启动过程中来寻找答案了。

在负责启动各种系统服务的ServerThread中，当核心系统服务启动完成后ServerThread会通过调用ActivityManagerService.systemReady()方法通知AMS系统已经就绪。这个systemReady()拥有一个名为goingCallback的Runnable实例作为参数。顾名思义，当AMS完成对systemReady()的处理后将会回调这一Runnable的run()方法。而在这一run()方法中可以找到SystemUI的身影：

```
[SystemServer.java-->ServerThread]
ActivityManagerService.self().systemReady(newRunnable() {
    publicvoid run() {
        // 调用startSystemUi()
        if(!headless) startSystemUi(contextF);
       ......
    }
}
```

进一步地，在startSystemUI()方法中：

```
[SystemServer.java-->ServerThread.startSystemUi()]
static final void startSystemUi(Context context) {
    Intentintent = new Intent();
    // 设置SystemUIService作为启动目标
   intent.setComponent(new ComponentName("com.android.systemui",
               "com.android.systemui.SystemUIService"));
    // 启动SystemUIService
   context.startServiceAsUser(intent, UserHandle.OWNER);
}
```

可见，当核心的系统服务启动完毕后，ServerThread通过Context.startServiceAsUser()方法完成了SystemUIService的启动。

2. SystemUIService的创建

参考SystemUIService的onCreate()的实现：

```
[SystemUIService.java-->SystemUIService.onCreate()]
/* **①SERVICES数组定义了运行于SystemUIService之中的子服务列表。**当SystemUIService服务启动
  时将会依次启动列表中所存储的子服务 */
final Object[] SERVICES = new Object[] {
        0,// 0号元素存储的其实是一个字符串资源号，这个字符串资源存储了实现了状态栏/导航栏的类名
       com.android.systemui.power.PowerUI.class,
       com.android.systemui.media.RingtonePlayer.class,
    };
public void onCreate() {
    ......
   IWindowManager wm = WindowManagerGlobal.getWindowManagerService();
    try {
        /* **② 根据IWindowManager.hasSystemNavBar()的返回值选择一个合适的**
**          状态栏与导航栏的实现** */
       SERVICES[0] = wm.hasSystemNavBar()
               ? R.string.config_systemBarComponent
               : R.string.config_statusBarComponent;
    } catch(RemoteException e) {......}
    finalint N = SERVICES.length;
    //mServices数组中存储了子服务的实例
   mServices = new SystemUI[N];
    for (inti=0; i&lt;N; i++) {
       Class cl = chooseClass(SERVICES[i]);
        try{
           // **③ 实例化子服务并将其存储在mServices数组中**
           mServices[i] = (SystemUI)cl.newInstance();
        }catch (IllegalAccessException ex) {......}
        // **④ 设置Context，并通过调用其start()方法运行它**
       mServices[i].mContext = this;
       mServices[i].start();
    }
}
```

除了onCreate()方法之外，SystemUIService没有其他有意义的代码了。显而易见，SystemUIService是一个容器。在其启动时，将会逐个实例化定义在SERVICIES列表中的继承自SystemUI抽象类的子服务。在调用了子服务的start()方法之后，SystemUIService便不再做任何其他的事情，任由各个子服务自行运行。而状态栏导航栏则是这些子服务中的一个。

值得注意的是，onCreate()方法根据IWindowManager.hasSystemNavBar()方法的返回值为状态栏/导航栏选择了不同的实现。进行这一选择的原因为了能够在大尺寸的设备中更有效地利用屏幕空间。在小屏幕设备如手机中，由于屏幕宽度有限，Android采取了状态栏与导航栏分

离的布局方案，也就是说导航栏与状态栏占用了更多的垂直空间，使得导航栏的虚拟按键尺寸足够大以及状态栏的信息量足够多。而在大屏幕设备如平板电脑中，由于屏幕宽度比较大，足以在一个屏幕宽度中同时显示足够大的虚拟按键以及足够多的状态栏信息量，此时可以选择将状态栏与导航栏功能集成在一起成为系统栏作为大屏幕下的布局方案，以节省对垂直空间的占用。

hasSystemNavBar()的返回值取决于PhoneWindowManager.mHasSystemNavBar成员的取值。因此在PhoneWindowManager.setInitialDisplaySize()方法中可以得知Android在两种布局方案中进行选择的策略。

```
[PhoneWindowManager.java-->PhoneWindowManager.setInitialDisplaySize()]
public void setInitialDisplaySize(Display display,int width
                                                  , intheight, int density) {
    ......
    // **① 计算屏幕短边的DP宽度**
    intshortSizeDp = shortSize * DisplayMetrics.DENSITY_DEFAULT / density;
    // **② 屏幕宽度在720dp以内时，使用分离的布局方案**
    if(shortSizeDp &lt; 600) {
        mHasSystemNavBar= false;
       mNavigationBarCanMove = true;
    } elseif (shortSizeDp &lt; 720) {
       mHasSystemNavBar = false;
       mNavigationBarCanMove = false;
    }
    ......
}
```

在SystemUI中，分离布局方案的实现者是PhoneStatusBar，而集成布局方案的实现者则是TabletStatusBar。二者的本质功能是一致的，即提供虚拟按键、显示通知信息等，区别仅在于布局的不同、以及由此所衍生出的定制行为而已。因此不难想到，它们是从同一个父类中继承出来的。这一父类的名字是BaseStatusBar。本章将主要介绍PhoneStatusBar的实现，读者可以类比地对TabletStatusBar进行研究。

7.1.2 状态栏与导航栏的创建

如7.1.1节所述，状态栏与导航栏的启动由其PhoneStatusBar.start()完成。参考其实现：

```
[PhoneStatusBar.java-->PhoneStatusBar.start()]
public void start() {
    ......
    // **① 调用父类BaseStatusBar的start()方法进行初始化。**
   super.start();
    // 创建导航栏的窗口
    addNavigationBar();
    // **② 创建PhoneStatusBarPolicy。**PhoneStatusBarPolicy定义了系统通知图标的设置策略
   mIconPolicy = new PhoneStatusBarPolicy(mContext);
}
```

参考BaseStatusBar.start()的实现，这段代码比较长，并且涉及到了本章随后会详细介绍的内容。因此倘若读者阅读起来比较吃力可以仅关注那三个关键步骤。在完成本章的学习之后再回过头来阅读这部分代码便会发现十分简单了。

```
[BaseStatusBar-->BaseStatusBar.start()]
public void start() {
    /* 由于状态栏的窗口不属于任何一个Activity，所以需要使用第6章所介绍的WindowManager
      进行窗口的创建 */
   mWindowManager = (WindowManager)mContext
                            .getSystemService(Context.WINDOW_SERVICE);
    /* 在第4章介绍窗口的布局时曾经提到状态栏的存在对窗口布局有着重要的影响。因此状态栏中
      所发生的变化有必要通知给WMS */
   mWindowManagerService = WindowManagerGlobal.getWindowManagerService();
    ......
    /*mProvisioningOberver是一个ContentObserver。
      它负责监听Settings.Global.DEVICE_PROVISIONED设置的变化。这一设置表示此设备是否已经
      归属于某一个用户。比如当用户打开一个新购买的设备时，初始化设置向导将会引导用户阅读使用条款、
      设置帐户等一系列的初始化操作。在初始化设置向导完成之前，
      Settings.Global.DEVICE_PROVISIONED的值为false，表示这台设备并未归属于某
      一个用户。
      当设备并未归属于某以用户时，状态栏会禁用一些功能以避免信息的泄露。mProvisioningObserver
      即是用来监听设备归属状态的变化，以禁用或启用某些功能 */
   mProvisioningObserver.onChange(false); // set up
   mContext.getContentResolver().registerContentObserver(
           Settings.Global.getUriFor(Settings.Global.DEVICE_PROVISIONED), true,
           mProvisioningObserver);
    /* **① 获取IStatusBarService的实例。**IStatusBarService是一个系统服务，由ServerThread
      启动并常驻system_server进程中。IStatusBarService为那些对状态栏感兴趣的其他系统服务定
      义了一系列API，然而对SystemUI而言，它更像是一个客户端。因为IStatusBarService会将操作
      状态栏的请求发送给SystemUI，并由后者完成请求 */
   mBarService = IStatusBarService.Stub.asInterface(
           ServiceManager.getService(Context.STATUS_BAR_SERVICE));
    /* 随后BaseStatusBar将自己注册到IStatusBarService之中。以此声明本实例才是状态栏的真正
      实现者，IStatusBarService会将其所接受到的请求转发给本实例。
      “天有不测风云”，SystemUI难免会因为某些原因使得其意外终止。而状态栏中所显示的信息并不属于状态
      栏自己，而是属于其他的应用程序或是其他的系统服务。因此当SystemUI重新启动时，便需要恢复其
      终止前所显示的信息以避免信息的丢失。为此，IStatusBarService中保存了所有的需要状态栏进行显
      示的信息的副本，并在新的状态栏实例启动后，这些副本将会伴随着注册的过程传递给状态栏并进行显示，
      从而避免了信息的丢失。
      从代码分析的角度来看，这一从IstatusBarService中取回信息副本的过程正好完整地体现了状态栏
      所能显示的信息的类型*/
    /*iconList是向IStatusBarService进行注册的参数之一。它保存了用于显示在状态栏的系统状态
      区中的状态图标列表。在完成注册之后，IStatusBarService将会在其中填充两个数组，一个字符串
      数组用于表示状态的名称，一个StatusBarIcon类型的数组用于存储需要显示的图标资源。
      关于系统状态区的工作原理将在7.2.3节介绍*/
   StatusBarIconList iconList = new StatusBarIconList();
    /*notificationKeys和StatusBarNotification则存储了需要显示在状态栏的通知区中通知信息。
      前者存储了一个用Binder表示的通知发送者的ID列表。而notifications则存储了通知列表。二者
      通过索引号一一对应。关于通知的工作原理将在7.2.2节介绍 */
   ArrayList&lt;IBinder&gt; notificationKeys = newArrayList&lt;IBinder&gt;();
   ArrayList&lt;StatusBarNotification&gt; notifications
                                 = newArrayList&lt;StatusBarNotification&gt;();
    /*mCommandQueue是CommandQueue类的一个实例。CommandQueue继承自IStatusBar.Stub。
      因此它是IStatusBar的Bn端。在完成注册后，这一Binder对象的Bp端将会保存在
     IStatusBarService之中。因此它是IStatusBarService与BaseStatusBar进行通信的桥梁。
      */
    mCommandQueue= new CommandQueue(this, iconList);
    /*switches则存储了一些杂项：禁用功能列表，SystemUIVisiblity，是否在导航栏中显示虚拟的
      菜单键，输入法窗口是否可见、输入法窗口是否消费BACK键、是否接入了实体键盘、实体键盘是否被启用。
      在后文中将会介绍它们的具体影响 */
    int[]switches = new int[7];
   ArrayList&lt;IBinder&gt; binders = new ArrayList&lt;IBinder&gt;();
    try {
        // **② 向IStatusBarServie进行注册，并获取所有保存在IStatusBarService中的信息副本**
       mBarService.registerStatusBar(mCommandQueue, iconList,
                                       notificationKeys,notifications,
                                      switches, binders);
    } catch(RemoteException ex) {......}
    // **③ 创建状态栏与导航栏的窗口。**由于创建状态栏与导航栏的窗口涉及到控件树的创建，因此它由子类
    PhoneStatusBar或TabletStatusBar实现，以根据不同的布局方案选择创建不同的窗口与控件树 */
   createAndAddWindows();
    /*应用来自IStatusBarService中所获取的信息
      mCommandQueue已经注册到IStatusBarService中，状态栏与导航栏的窗口与控件树也都创建完毕
      因此接下来的任务就是应用从IStatusBarService中所获取的信息 */
   disable(switches[0]); // 禁用某些功能
   setSystemUiVisibility(switches[1], 0xffffffff); // 设置SystemUIVisibility
```

```
        topAppWindowChanged(switches[2]!= 0); // 设置菜单键的可见性
        // 根据输入法窗口的可见性调整导航栏的样式
       setImeWindowStatus(binders.get(0), switches[3], switches[4]);
        // 设置硬件键盘信息。
       setHardKeyboardStatus(switches[5] != 0, switches[6] != 0);
        // 依次向系统状态区添加状态图标
        int N = iconList.size();
        ......
        // 依次向通知栏添加通知
        N = notificationKeys.size();
        ......
        /* 至此，与IStatusBarService的连接已建立，状态栏与导航栏的窗口也已完成创建与显示，并且
          保存在IStatusBarService中的信息都已完成了显示或设置。状态栏与导航栏的启动正式完成 */
    }
```

可见，状态栏与导航栏的启动分为如下几个过程：

- 获取IStatusBarService，IStatusBarService是运行于system_server的一个系统服务，它接受操作状态栏/导航栏的请求并将其转发给BaseStatusBar。为了保证SystemUI意外退出后不会发生信息丢失，IStatusBarService保存了所有需要状态栏与导航栏进行显示或处理的信息副本。
- 将一个继承自IStatusBar.Stub的CommandQueue的实例注册到IStatusBarService以建立通信，并将信息副本取回。
- 通过调用子类的createAndAddWindows()方法完成状态栏与导航栏的控件树及窗口的创建与显示。
- 使用从IStatusBarService取回的信息副本。

7.1.3 理解IStatusBarService

那么IStatusBarService的真身如何呢？它的实现者是StatusBarManagerService。由于状态栏导航栏与它的关系十分密切，因此需要对其有所了解。

与WindowManagerService、InputManagerService等系统服务一样，StatusBarManagerService在ServerThread中创建。参考如下代码：

```
[SystemServer.java-->ServerThread.run()]
public void run() {
    try {
        /* 创建一个StatusBarManagerService的实例，并注册到ServiceManager中使其成为
          一个系统服务 */
       statusBar = new StatusBarManagerService(context, wm);
       ServiceManager.addService(Context.STATUS_BAR_SERVICE, statusBar);
    } catch(Throwable e) {......}
}
再看其构造函数：
[StatusBarManagerService.java-->StatusBarManagerService.StatusBarManagerService()]
public StatusBarManagerService(Context context,WindowManagerService windowManager) {
    mContext= context;
   mWindowManager = windowManager;
    // 监听实体键盘的状态变化
   mWindowManager.setOnHardKeyboardStatusChangeListener(this);
    // 初始化状态栏的系统状态区的状态图标列表。关于系统状态区的工作原理将在7.2.3节介绍
    finalResources res = context.getResources();
    mIcons.defineSlots(res.getStringArray(
                            com.android.internal.R.array.config_statusBarIcons));
}
```

这基本上是系统服务中最简单的构造函数了，在这里并没有发现能够揭示StatusBarManagerService的工作原理的线索（由此也可以预见StatusBarManagerService的实现十分简单）。

接下来参考StatusBarManagerService.registerStatusBar()的实现。这个方法由SystemUI中的BaseStatusBar调用，用于建立其与StatusBarManagerService的通信连接，并取回保存在其中的信息副本。

```
[StatusBarManagerService.java-->StatusBarManagerService.registerStatusBar()]
public void registerStatusBar(IStatusBar bar,StatusBarIconList iconList,
        List&lt;IBinder&gt; notificationKeys,List&lt;StatusBarNotification&gt; notificati
        intswitches[], List&lt;IBinder&gt; binders) {
    /* 首先是权限检查。状态栏与导航栏是Android系统中一个十分重要的组件，因此必须避免其他应用
      调用此方法对状态栏与导航栏进行偷梁换柱。因此要求方法的调用者必须具有一个签名级的权限
       android.permission.STATUS_BAR_SERVICE*/
   enforceStatusBarService();
    /* **① 将bar参数保存到mBar成员中。**bar的类型是IStatusBar，它即是BaseStatusBar中的
     CommandQueue的Bp端。从此之后，StatusBarManagerService将通过mBar与BaseStatusBar
      进行通信。因此可以理解mBar就是SystemUI中的状态栏与导航栏 */
    mBar =bar;
    // **② 接下来依次为调用者返回信息副本**
    // 系统状态区的图标列表
   synchronized (mIcons) { iconList.copyFrom(mIcons); }
    // 通知区的通知信息
   synchronized (mNotifications) {
        for(Map.Entry&lt;IBinder,StatusBarNotification&gt; e: mNotifications.entrySet())
           notificationKeys.add(e.getKey());
           notifications.add(e.getValue());
        }
    }
    //switches中的杂项
   synchronized (mLock) {
       switches[0] = gatherDisableActionsLocked(mCurrentUserId);
        ......
    }
    ......
}
```

可见StatusBarManagerService.registerStatusBar()的实现也十分简单。主要是保存BaseStatusBar中的CommandQueue的Bp端到mBar成员之中，然后再把信息副本填充到参数里去。尽管简单，但是从其实现中可以预料到StatusBarManagerService的工作方式：当它接受到操作状态栏与导航栏的请求时，首先将请求信息保存到副本之中，然后再将这一请求通过mBar发送给BaseStatusBar。以设置系统状态区图标这一操作为例，参考如下代码：

```
[StatusBarManagerService.java-->StatusBarManagerService.setIcon()]
public void setIcon(String slot, StringiconPackage, int iconId, int iconLevel,
       String contentDescription) {
    /* 首先一样是权限检查，与registerStatusBar()不同，这次要求的是一个系统级别的权限
      android.permission.STATUS_BAR。因为设置系统状态区图标的操作不允许普通应用程序进行。
      其他的操作诸如添加一条通知则不需要此权限 */
   enforceStatusBar();
   synchronized (mIcons) {
        intindex = mIcons.getSlotIndex(slot);
        ......
       StatusBarIcon icon = new StatusBarIcon(iconPackage, UserHandle.OWNER,iconId,
               iconLevel, 0,
               contentDescription);
        // **① 将图标信息保存在副本之中**
       mIcons.setIcon(index, icon);
        // **② 将设置请求发送给BaseStatusBar**
        if(mBar != null) {
           try {
               mBar.setIcon(index, icon);
           } catch (RemoteException ex) {......}
        }
    }
}
```

纵观StatusBarManagerService中的其他方法，会发现它们与setIcon()方法的实现十分类似。从而可以得知StatusBarManagerService的作用与工作原理如下：

- 它是SystemUI中的状态栏与导航栏在system_server中的代理。所有对状态栏或导航来有需求的对象都可以通过获取StatusBarManagerService的实例或Bp端达到其目的。只不过使用者必须拥有能够完成操作的相应权限。
- 它保存了状态栏／导航栏所需的信息副本，用于在SystemUI意外退出之后的恢复。

7.1.4 SystemUI的体系结构

完成了对SystemUI的启动过程的分析之后便可以对其体系结构做出总结，如图7-1所示。

- SystemUIService，一个普通的Android服务，它以一个容器的角色运行于SystemUI进程中。在它内部运行着多个子服务，其中之一便是状态栏与导航栏的实现者——BaseStatusBar的子类之一。
- IStatusBarService，即系统服务StatusBarManagerService是状态栏导航栏向外界提供服务的前端接口，运行于system_server进程中。
- BaseStatusBar及其子类是状态栏与导航栏的实际实现者，运行于SystemUIService中。

- IStatusBar，即SystemUI中的CommandQueue是联系StatusBarManagerService与BaseStatusBar的桥梁。
- SystemUI中还包含了ImageWallpaper、RecentPanel以及TakeScreenshotService等功能的实现。它们是Service、Activity等标准的Android应用程序组件，并且互相独立。对这些功能感兴趣的使用者可以通过startService()/startActivity()等方式方便地启动相应的功能。

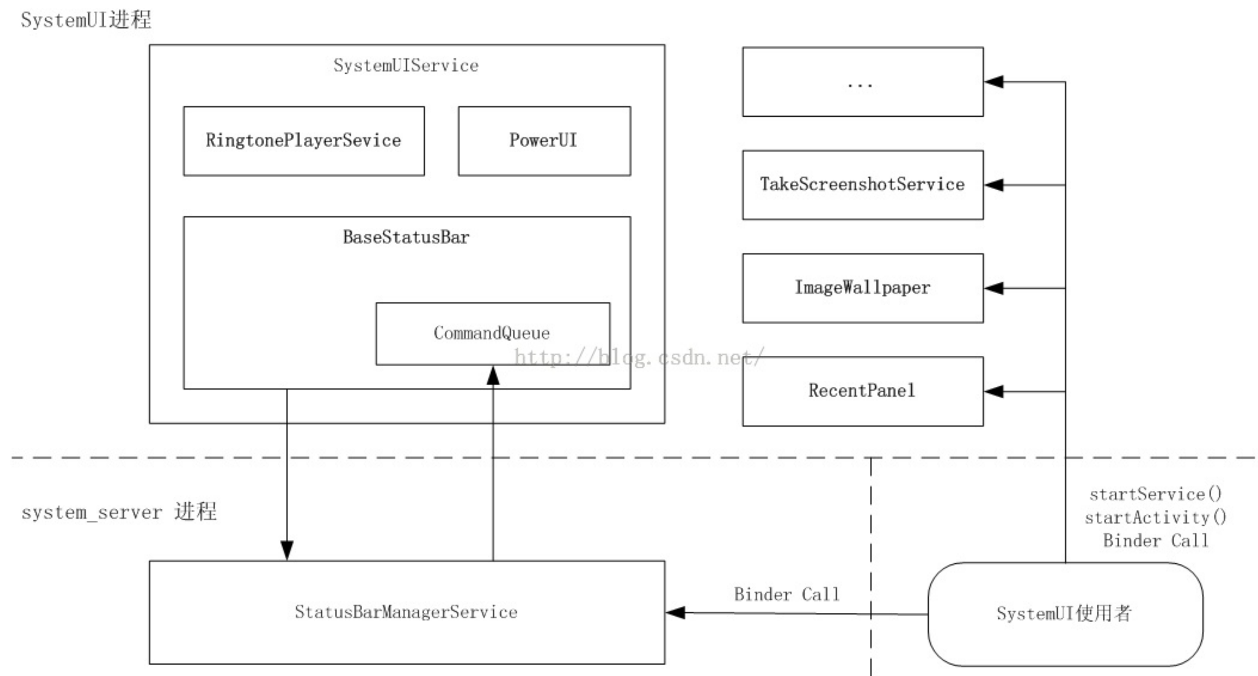


图 7 - 1 SystemUI的体系结构

在本章将主要介绍SystemUI中最常用的状态栏、导航栏以及RecentPanel的实现。ImageWallpaper将在第8章中进行详细地介绍。而SystemUI其他的功能读者可以自行研究。

7.2 深入理解状态栏

如7.1.1节所述，SystemUI中存在两种状态栏与导航栏的实现——即状态栏与导航栏分离的布局的PhoneStatusBar以及状态栏与导航栏集成布局的TabletStatusBar两种。除了布局差异之外，二者并无本质上的差别，因此本节将主要介绍PhoneStatusBar下的状态栏的实现。

作为一个将所有信息集中显示的场所，状态栏对需要显示的信息做了以下的五个分类。

- 通知信息：它可以在状态栏左侧显示一个图标以引起用户的主意，并在下拉卷帘中为用户显示更加详细的信息。这是状态栏所能提供的信息显示服务之中最灵活的一种功能。它对信息种类以及来源没有做任何限制。使用者可以通过StatusBarManagerService所提供的接口向状态栏中添加或移除一条通知信息。

- 时间信息：显示在状态栏最右侧的一个小型数字时钟，是一个名为Clock的继承自TextView的控件。它监听了几个和时间相关的广播：ACTION_TIME_TICK、ACTION_TIME_CHANGED、ACTION_TIMEZONE_CHANGED以及ACTION_CONFIGURATION_CHANGED。当其中一个广播到来时从Calendar类中获取当前的系统时间，然后进行字符串格式化后显示出来。时间信息的维护工作在状态栏内部完成，因此外界无法通过API修改时间信息的显示或行为。

- 电量信息：显示在数字时钟左侧的一个电池图标，用于提示设备当前的电量情况。它是一个被BatteryController类所管理的ImageView。BatteryController通过监听android.intent.action.BATTERY_CHANGED广播以从BetteryService中获取电量信息，并根据电量信息选择一个合适的电池图标显示在ImageView上。同时间信息一样，这也是在状态栏内部维护的，外界无法干预状态栏对电量信息的显示行为。

- 信号信息：显示在电量信息的左侧的一系列ImageView，用于显示系统当前的Wifi、移动网络的信号状态。用户所看到的Wifi图标、手机信号图标、飞行模式图标都属于信号信息的范畴。它们被NetworkController类维护着。NetworkController监听了一系列与信号相关的广播如WIFI_STATE_CHANGED_ACTION、ACTION_SIM_STATE_CHANGED、ACTION_AIRPLANE_MODE_CHANGED等，并在这些广播到来时显示、更改或移除相关的ImageView。同样，外界无法干预状态栏对信号信息的显示行为。

- 系统状态图标区：这个区域用一系列图标标识系统当前的状态，位于信号信息的左侧，与状态栏左侧通知信息隔岸相望。通知信息类似，StatusBarManagerService通过setIcon()接口为外界提供了修改系统状态图标区的图标的途径，而然它对信息的内容有很强的限制。首先，系统状态图标区无法显示图标以外的信息，另外，系统状态图标区的对其所显示的图标数量以及图标所表示的意图有着严格的限制。

由于时间信息、电量信息以及信号信息的实现原理比较简单而且与状态栏外界相对隔离，因此读者可以通过分析上文所介绍的相关组件自行研究。本节将主要介绍状态栏的一下几个方面的内容：

- 状态栏窗口的创建与控件树结构。
- 通知的管理与显示。
- 系统状态图标区的管理与显示。

7.2.1 状态栏窗口的创建与控件树结构

1. 状态栏窗口的创建

在7.1.2节所引用的BaseStatusBar.start()方法的代码中调用了createAndAddWindows()方法进行状态栏窗口的创建。很显然，createAndAddWindow()由PhoneStatusBar或TabletStatusBar实现。以PhoneStatusBar为例，参考其代码：

```
[PhoneStatusBar.java-->PhoneStatusBar.createAndAddWindow()]
public void createAndAddWindows() {
   addStatusBarWindow(); // 直接调用addStatusBarWindow()方法
}
```

在addStatusBarWindow()方法中，PhoneStatusBar将会构建状态栏的控件树并通过WindowManager的接口为其创建窗口。

```
[PhoneStatusBar.java-->PhoneStatusBar.addStatusBarWindow()]
private void addStatusBarWindow() {
    // **① 通过getStatusBarHeight()方法获取状态栏的高度**
    finalint height = getStatusBarHeight();
    // **② 为状态栏创建WindowManager.LayoutParams**
    finalWindowManager.LayoutParams lp = new WindowManager.LayoutParams(
           ViewGroup.LayoutParams.MATCH_PARENT, // 状态栏的宽度为充满整个屏幕宽度
           height, // 高度来自于getStatusBarHeight()方法
           WindowManager.LayoutParams.TYPE_STATUS_BAR, // 窗口类型
           WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE // 状态栏不接受按键事件
                 /* FLAG_TOUCHABLE_WHEN_WAKING这一标记将使得状态栏接受导致设备唤醒的触摸
                   事件。通常这一事件会在interceptMotionBeforeQueueing()的过程中被用于
                   唤醒设备（或从变暗状态下恢复），而InputDispatcher会阻止这一事件发送给
                   窗口。*/
               | WindowManager.LayoutParams.FLAG_TOUCHABLE_WHEN_WAKING
                  // FLAG_SPLIT_TOUCH允许状态栏支持触摸事件序列的拆分
               | WindowManager.LayoutParams.FLAG_SPLIT_TOUCH,
           PixelFormat.TRANSLUCENT); // 状态栏的Surface像素格式为支持透明度
    // 启用硬件加速
    lp.flags|= WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED;
    //StatusBar的gravity是LEFT和FILL_HORIZONTAL
   lp.gravity = getStatusBarGravity();
   lp.setTitle("StatusBar");
   lp.packageName = mContext.getPackageName();
    // **③ 创建状态栏的控件树**
   makeStatusBarView();
    // **④ 通过WindowManager.addView()创建状态栏的窗口**
   mWindowManager.addView(mStatusBarWindow, lp);
}
```

此方法提供了很多重要的信息。

首先是状态栏的高度，由getStatusBarHeight()从资源com.android.internal.R.dimen.status_bar_height中获得。这一资源定义在frameworks\base\core\res\res\values\dimens.xml中，默认为25dip。此资源同样在PhoneWindowManager中被用来计算作为布局准绳的八个矩形。

然后是状态栏窗口的LayoutParams的创建。LayoutParams描述了状态栏是怎样的一个窗口。TYPE_STATUS_BAR使得PhoneWindowManager为状态栏的窗口分配了较大的layer值，使其可以显示在其他应用窗口之上。FLAG_NOT_FOCUSABLE、FLAG_TOUCHABLE_WHEN_WAKING和FLAG_SPLIT_TOUCH则定义了状态栏对输入事件的响应行为。

注意 通过创建窗口所使用的LayoutParams来推断一个窗口的行为十分重要。在分析一个需要创建窗口的模块的工作原理时，从窗口创建过程往往是一个不错的切入点。

另外需要知道的是，窗口创建之后，其LayoutParams是会发生变化的。以状态栏为例，创建窗口时其高度为25dip，flags描述其不可接收按键事件。不过当用户按下状态栏导致卷帘下拉时，PhoneStatusBar会通过WindowManager.updateViewLayout()方法修改窗口的LayoutParams的高度为MATCH_PARENT，即充满整个屏幕以使得卷帘可以满屏显示，并且移除FLAG_NOT_FOCUSABLE，使得PhoneStatusBar可以通过监听BACK键以收回卷帘。

在makeStatusBarView()完成控件树的创建之后，WindowManager.addView()将根据控件树创建出状态栏的窗口。显而易见，状态栏控件树的根控件被保存在mStatusBarWindow成员中。

createStatusBarView()负责从R.layout.super_status_bar所描述的布局中实例化出一棵控件树。并从这个控件树中取出一些比较重要的控件并保存在对应的成员变量中。因此从R.layout.super_status_bar入手可以很容易地得知状态栏的控件树的结构：

2.状态栏控件树的结构

参考SystemUI下super_status_bar.xml所描述的布局内容，可以看到其根控件是一个名为StatusBarWindowView的控件，它继承自FrameLayout。在其下的两个直接子控件如下：

- @layout/status_bar所描述的布局。这是用户平时所见的状态栏。
- PenelHolder：这个继承自FrameLayout的控件是状态栏的卷帘。在其下的两个直接子控件@layout/status_bar_expanded以及@layout/quick_settings分别对应于卷帘之中的通知列表面板以及快速设定面板。

在正常情况下，StatusBarWindowView中只有@layout/status_bar所描述的布局是可见的，并且状态栏窗口为com.android.internal.R.dimen.status_bar_height所定义的高度。当StatusBarWindowView截获了ACTION_DOWN的触摸事件后，会修改窗口的高度为MATCH_PARENT，然后将PenelHolder设为可见并跟随用户的触摸轨迹，由此实现了卷帘的下拉效果。

说明 PenelHolder集成自FrameLayout。那么它如何做到在@layout/status_bar_expanded以及@layout/quick_settings两个控件之间进行切换显示呢？答案就在第6章所介绍的ViewGroup. getChildDrawingOrder()方法中。此方法的返回值影响了子控件的绘制顺序，同时也影响了控件接收触摸事件的优先级。当PenelHolder希望显示@layout/status_bar_expanded面版时，它在此方法中将此面版的绘制顺序放在最后，使其在绘制时能够覆盖@layout/quick_settings，并且优先接受触摸事件。反之则将@layout/quick_settings的绘制顺序放在最后即可。

因此状态栏控件树的第一层结构如图7-2所示。

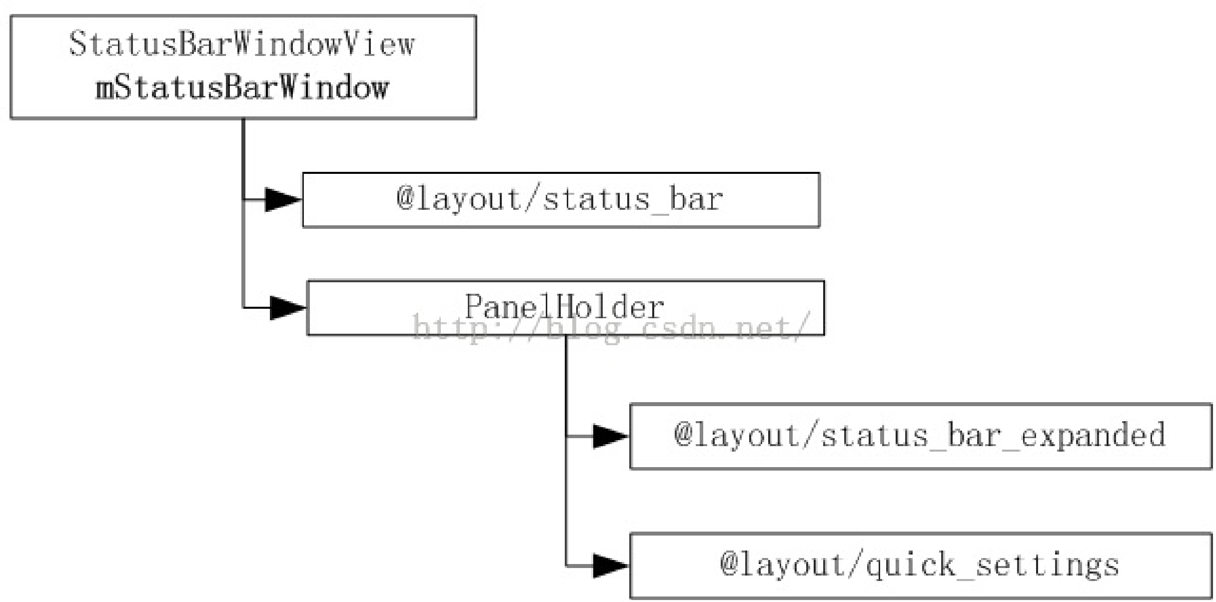


图 7 - 2状态栏控件树的结构1

再看status_bar.xml所描述的布局内容，其根控件是一个继承自FrameLayout的名为StatusBarView类型的控件，makeStatusBarView()方法会将其保存为mStatusBarView。其直接子控件有三个：

- @id/notification_lights_out，一个ImageView，并且一般情况下它是不可见的。在SystemUIVisiblity中有一个名为SYSTEM_UI_FLAG_LOW_PROFILE的标记。当一个应用程序希望让用户的注意力更多地集中在它所显示的内容时，可以在其SystemUIVisibility中添加这一标记。SYSTEM_UI_FLAG_LOW_PROFILE会使得状态栏与导航栏进入低辨识度模式。低辨识度模式下的状态栏将不会显示任何信息，只是在黑色背景中显示一个灰色圆点而已。而这一个黑色圆点即是这里的id/notification_lights_out。
- @id/status_bar_contents，一个LinearLayout，状态栏上各种信息的显示场所。
- @id/ticker，一个LinearLayout，其中包含了一个ImageSwitcher和一个TickerView。在正常情况下@id/ticker是不可见的。当一个新的通知到来时（例如一条新的短信），状态栏上会以动画方式逐行显示通知的内容，使得用户可以在无需下拉卷帘的情况下了解新通知的内容。这一功能在状态栏中被称之为Ticker。而@id/ticker则是完成Ticker功能的场所。makeStatusBarView()会将@id/ticker保存为mTickerView。

至此，状态栏控件树的结构可以扩充为图7-3所示。

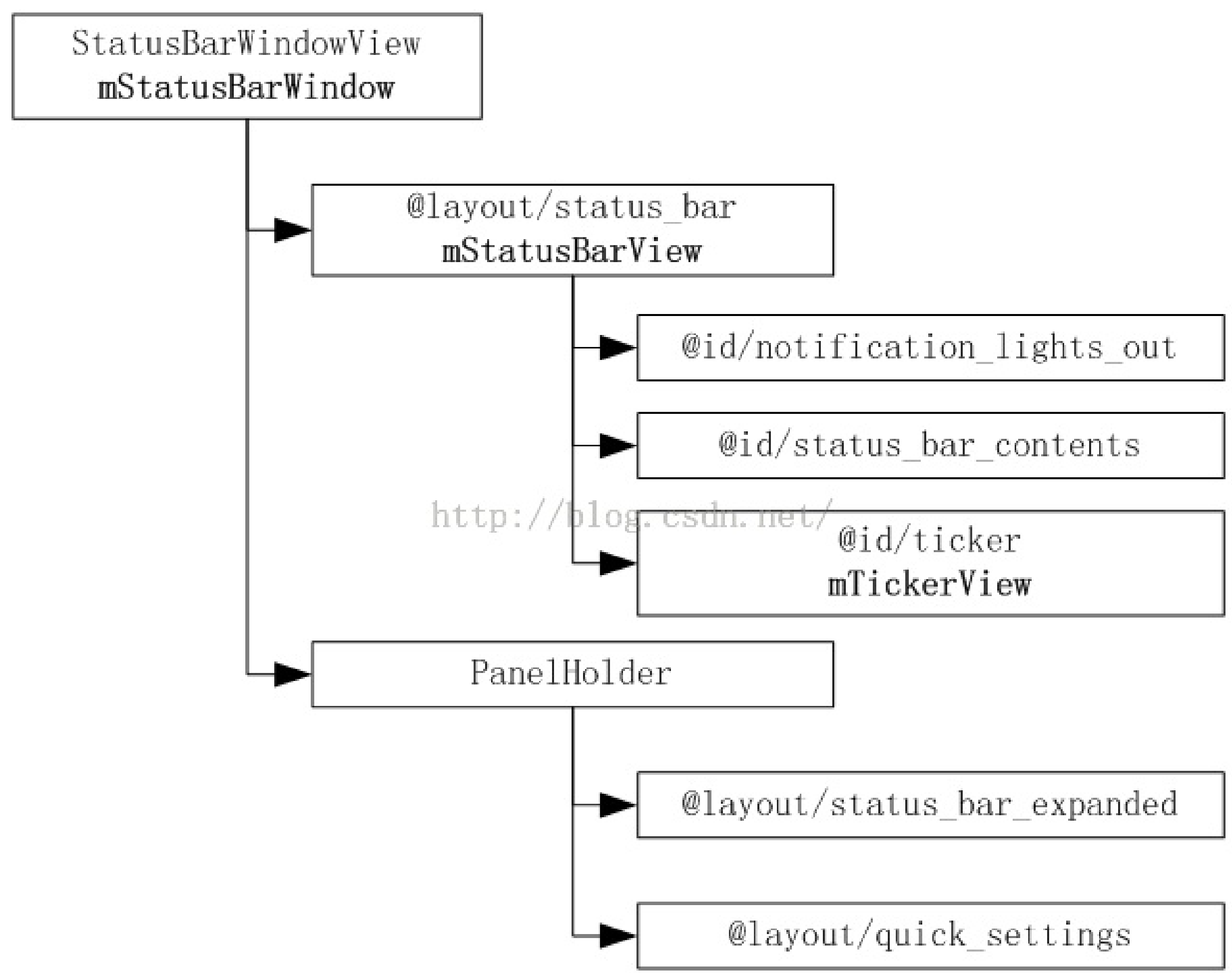


图 7 - 3状态栏控件树的结构2

再来分析@id/status_bar_contents所包含的内容。如前文所述，状态栏所显示的信息共有5种，因此@id/status_bar_contents中的子控件分别用来显示这5种信息。其中通知信息显示在@id/notification_icon_area里，而其他四种信息则显示在@id/system_icon_area之中。

- @id/notification_icon_area，一个LinearLayout。包含了两个子控件分别是类型为StatusBarIconView的@id/moreIcon以及一个类型为IconMerger的@id/notificationIcons。IconMerger继承自LinearLayout。通知信息的图标都会以一个StatusBarIconView的形式存储在IconMerger之中。而IconMeger和LinearLayout的区别在于，如果它在onLayout()的过程中发现会其内部所容纳的StatusBarIconView的总宽度超过了它自身的宽度，则会设置@id/moreIcon为可见，使得用户得知有部分通知图标因为显示空间不够而被隐藏。makeStausBarView()会将@id/notificationIcons保存为成员变量mNotificationIcons。因此当新的通知到来时，只要将一个StatusBarIconView放置到mNotificationIcons即可显示此通知的图标了。

- @id/system_icon_area，也是一个LinearLayout。它容纳了除通知信息的图标以外的四种信息的显示。在其中有负责显示时间信息的@id/clock，负责显示电量信息的@id/battery，负责信号信息显示的@id/signal_cluster以及负责容纳系统状态区图标的一

个LinearLayout——@id/statusIcons。其中@id/statusIcons会被保存到成员变量mStatusIcons中，当需要显示某一个系统状态图标时，将图标放置到mStatusIcons中即可。

注意 @id/system_icon_area的宽度定义为WRAP_CONTENT，而@id/notification_icon_area的weight被设置为1。在这种情况下，@id/system_icon_area将在状态栏右侧根据其所显示的图标个数调整其尺寸。而@id/notification_icon_area则会占用状态栏左侧的剩余空间。这说明了一个问题：系统图标区将优先占用状态栏的空间进行信息的显示。这也是IconMerger类以及@id/moreIcon存在的原因。

于是可以将图7-3扩展为图7-4。

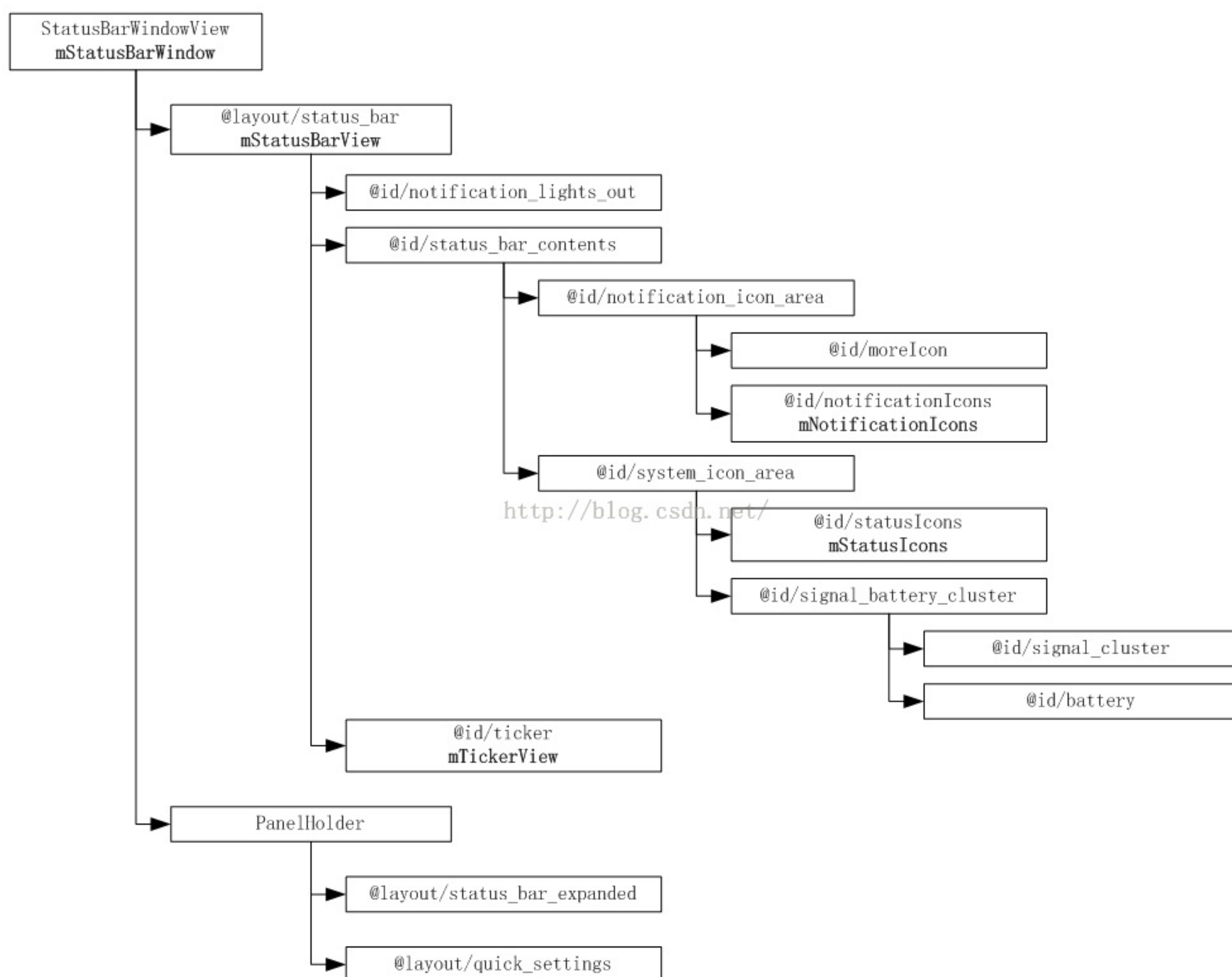


图 7 - 4状态栏控件树的结构3

另外，在@layout/status_bar_expanded之中有一个类型为NotificationRowLayout的控件@id/latestItems，并且会被makeStatusBarView()保存到mPile成员变量中。它位于下拉卷帘中，是通知信息列表的容器。

在分析控件树结构的过程中发现了如下几个重要的控件：

- mStatusBarWindow，整个状态栏的根控件。它包含了两棵子控件树，分别是常态下的状态栏以及下拉卷帘。

- mStatusBarView，常态下的状态栏。它所包含的三棵子控件树分别对应了状态栏的三种工作状态——低辨识度模式、Ticker以及常态。这三棵控件树会随着这三种工作状态的切换交替显示。
- mNotificationIcons，继承自LinearLayout的IconMerger控件的实例，负责容纳通知图标。当mNotificationIcons的宽度不足以容纳所有通知图标时，会将@id/moreIcon设置为可见以告知用户存在未显示的通知图标。
- mTickerView，实现了当新通知到来时的动画效果，使得用户可以在无需下拉卷帘的情况下了解新通知的内容。
- mStatusIcons，一个LinearLayout，它是系统状态图标区，负责容纳系统状态图标。
- mPile，一个NotificationRowLayout，它作为通知列表的容器被保存在下拉卷帘中。因此当一个通知信息除了需要将其图标添加到mNotificationIcons以外，还需要将其详细信息（标题、描述等）添加到mPile中，使得用户在下来卷帘中可以看到它。

对状态栏控件树的结构分析至此便告一段落了。接下来将从通知信息以及系统状态图标两个方面介绍状态栏的工作原理。希望读者能够理解本节所介绍的几个重要控件所在的位置以及其基本功能，这将使得后续内容的学习更加轻松。

7.2.2 通知信息的管理与显示

通知信息是状态栏中最常用的功能之一。根据用户是否拉下下拉卷帘，通知信息表现为一个位于状态栏的图标，或在下拉卷帘中的一个条目。另外，通知信息还可以在其添加入状态栏之时发出声音，以提醒用户注意查看。通知信息即可以表示一条事件，如新的短消息到来、出现了一条未接来电等，也可以用来表示一个正在后台持续进行着的工作，如正在下载某一文件、正在播放音乐等。

1.通知信息的发送

任何使用者都可以通过NotificationManager所提供的接口向状态栏添加一则通知信息。通知信息的详细内容可以通过一个Notification类的实例来描述。

Notification类中包含如下几个用于描述通知信息的关键字段。

- icon，一个用于描述一个图标的资源id，用于显示在状态栏之上。每条通知信息必须提供一个有效的图标资源，否则此信息将会被忽略。
- iconLevel，如果icon所描述的图标资源存在level，那么iconLevel则用于告知状态栏将显示图标资源的那一个level。
- number，一个int型变量用于表示通知数目。例如，当有3条新的短信时，没有必要使用三个通知，而是将一个通知的number成员设置为3，状态栏会将这一数字显示在通知图标上。

- contentIntent，一个PendingIntent的实例，用于告知状态栏当在下拉卷帘中点击本条通知时应当执行的动作。contentIntent往往用于启动一个Activity以便让用户能够查看关于此条通知的详细信息。例如，当用户点击一条提示新短信的通知时，短信应用将会被启动并显示短信的详细内容。
- deleteIntent，一个PendingIntent的实例，用于告知状态栏当用户从下拉卷帘中删除本条通知时应当执行的动作。deleteIntent往往用在表示某个工作正在后台进行的通知中，以便当用户从下拉卷帘中删除通知时，发送者可以终止此后台工作。
- tickerText，一条文本。当通知信息被添加时，状态栏将会在其上逐行显示这条信息。其目的在于使用户无需进行卷帘下拉操作即可从快速获取通知的内容。
- fullScreenIntent，一个PendingIntent的实例，用于告知状态栏当此条信息被添加时应当执行的动作，一般这一动作是启动一个Activity用于显示与通知相关的详细信息。fullScreenIntent其实是一个替代tickerText的设置。当Notification中指定了fullScreenIntent时，StatusBar将会忽略tickerText的设置。因为这两个设置的目的都是为了让用户可以在第一时间了解通知的内容。不过相对于tickerText，fullScreenIntent强制性要明显得多，因为它将打断用户当前正在进行的工作。因此fullScreenIntent应该仅用于通知非常重要或紧急的事件，比如说来电或闹钟。
- contentView/bigContentView，RemoteView的实例，可以用来定制通知信息在下拉卷帘中的显示形式。一般来讲，相对于contentView，bigContentView可以占用更多空间以显示更加详细的内容。状态栏将根据自己的判断选择将通知信息显示为contentView或是bigContentView。
- sound与audioStreamType，指定一个用于播放通知声音的Uri及其所使用的音频流类型。在默认情况下，播放通知声音所用的音频流类型为STREAM_NOTIFICATION。
- vibrate，一个float数组，用于描述震动方式。
- ledARGB/ledOnMS/ledOffMS，指定当此通知被添加到状态栏时设备上的LED指示灯的行为，这几个设置需要硬件设备的支持。
- defaults，用于指示声音、震动以及LED指示灯是否使用系统的默认行为。
- flags，用于存储一系列用于定制通知信息行为的标记。通知信息的发送者可以根据需求在其中加入这样的标记：FLAG_SHOW_LIGHTS要求使用LED指示灯，FLAG_ONGOING_EVENT指示通知信息用于描述一个正在进行的后台工作，FLAG_INSISTENT指示通知声音将持续播放直到通知信息被移除或被用户查看，FLAG_ONLY_ARLERT_ONCE指示任何时候通知信息被加入到状态栏时都会播放一次通知声音，FLAG_AUTO_CANCEL指示当用户在下拉卷帘中点击通知信息时自动将其移出，FLAG_FOREGROUND_SERVICE指示此通知用来表示一个正在以foreground形式运行的服务。

- priority，描述了通知的重要性级别。通知信息的级别从低到高共分为MIN(-2)、LOW(-1)、DEFAULT(0)以及HIGH(1)四级。低优先级的通知信息有可能不会被显示给用户，或显示在通知列表中靠下的位置。

在随后的讨论中将会详细介绍这些信息如何影响通知信息的显示与行为。

当通知信息的发送者根据需求完成了Notification实例的创建之后，便可以通过NotificationManager.notify()方法将通知显示在状态栏上。

notify()方法要求通知信息的发送者除了提供一个Notification实例之外，还需要提供一个字符串类型的参数tag，以及int类型的参数id，这两个参数一并确定了信息的意图。当一条通知信息已经被提交给NotificationManager.notify()并且仍然显示在状态栏中时，它将会被新提交的拥有相同意图（即相同的tag以及相同的id）通知信息所替换。

参考NotificationManager.notify()方法的实现：

```
[NotificationManager.java-->NotificationManager.notify()]
public void notify(String tag, int id,Notification notification)
{
    int[]idOut = new int[1];
    // **① 获取NotificationManagerService的Bp端代理**
   INotificationManager service = getService();
    // **② 获取信息发送者的包名**
    Stringpkg = mContext.getPackageName();
    ......
    try {
        // **③ 将包名、tag、id以及Notification实例一并提交给NotificationManagerService**
       service.enqueueNotificationWithTag(pkg, tag, id, notification, idOut,
               UserHandle.myUserId());
    } catch(RemoteException e) {......}
}
```

NotificationManager会将通知信息发送给NotificationManagerService，并由NotificationManagerService对信息进行进一步处理。注意Notification将通知发送者的包名作为参数传递给了NotificationManagerService。对于一个应用程序来说，tag与id而这一起确定了通知的意图。由于NotificationManagerService作为一个系统服务需要接受来自各个应用程序通知信息，因此对NotificationManagerService来说，确定通知的意图需要在tag与id之外再增加一项：通知发送者的包名。因此由于包名的不一样，来自两个应用程序的具有相同tag与id的通知信息之间不会发生任何冲突。另外将包名作为通知意图的元素之一的原因出于对信息安全考虑。

而将一则通知信息从状态栏中移除则简单得多了，NotificationManager.cancel()方法可以提供这一操作，它接受tag、id作为参数用于指明希望移除的通知所具有的意图。

第8章 深入理解Android壁纸（节选）

本章主要内容：

- 讨论动态壁纸的实现。
- 在动态壁纸的基础上讨论静态壁纸的实现。
- 讨论WMS对壁纸窗口所做的特殊处理。

本章涉及的源代码文件名及位置：

- WallpaperManagerService.java

frameworks/base/services/java/com/android/server/WallpaperManagerService.java

- WallpaperService.java

frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

- ImageWallpaper.java

frameworks/base/packages/SystemUI/src/com/android/systemui/ImageWallpaper.java

- WallpaperManager.java

frameworks/base/core/java/android/app/WallpaperManager.java

- WindowManagerService.java

frameworks/base/services/java/com/android/server/wm/WindowManagerService.java

- WindowStateAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowStateAnimator.java

- WindowAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowAnimator.java

8.1 初识Android壁纸

本章将对壁纸的实现原理进行讨论。在Android中，壁纸分为静态与动态两种。静态壁纸是一张图片，而动态壁纸则以动画为表现形式，或者可以对用户的操作作出反应。这两种形式看似差异很大，其实二者的本质是统一的。它们都以一个Service的形式运行在系统后台，并在

一个类型为TYPE_WALLPAPER的窗口上绘制内容。进一步讲，静态壁纸是一种特殊的动态壁纸，它仅在窗口上渲染一张图片，并且不会对用户的操作作出反应。因此本章将首先通过动态壁纸的实现讨论Android壁纸的实现与管理原理，然后在对静态壁纸的实现做介绍。

Android壁纸的实现与管理分为三个层次：

- WallpaperService与Engine。同SystemUI一样，壁纸运行在一个Android服务之中，这个服务的名字叫做WallpaperService。当用户选择了一个壁纸之后，此壁纸所对应的WallpaperService便会启动并开始进行壁纸的绘制工作，因此继承并定制WallpaperService是开发者进行壁纸开发的第一步。Engine是WallpaperService中的一个内部类，实现了壁纸窗口的创建以及Surface的维护工作。另外，Engine提供了可供子类重写的一系列回调，用于通知壁纸开发者关于壁纸的生命周期、Surface状态的变化以及对用户的输入事件进行响应。可以说，Engine类是壁纸实现的核心所在。壁纸开发者需要继承Engine类，并重写其提供的回调以完成壁纸的开发。这一层次的内容主要体现了壁纸的实现原理。

- WallpaperManagerService，这个系统服务用于管理壁纸的运行与切换，并通过WallpaperManager类向外界提供操作壁纸的接口。当通过WallpaperManagaer的接口进行壁纸的切换时，WallpaperManagerService会取消当前壁纸的WallpaperService的绑定，并启动新壁纸的WallpaperService。另外，Engine类进行窗口创建时所使用的窗口令牌也是由WallpaperManagerService提供的。这一层次主要体现了Android对壁纸的管理方式。

- WindowManagerService，用于计算壁纸窗口的Z序、可见性以及为壁纸应用窗口动画。壁纸窗口(TYPE_WALLPAPER)的Z序计算不同于其他类型的窗口。其他窗口依照其类型会有固定的mBaseLayer以及mSubLayer，并结合它们所属的Activity的顺序或创建顺序进行Z序的计算，因此这些窗口的Z序相对固定。而壁纸窗口则不然，它的Z序会根据FLAG_SHOW_WALLPAPER标记在其它窗口的LayoutParams.flags中的存在情况而不断地被调整。这一层次主要体现了Android对壁纸窗口的管理方式。

本章将通过对动态壁纸切换的过程进行分析揭示WallpaperService、Engine以及WallpaperManagerService三者的实现原理以及协作情况。静态壁纸作为动态壁纸的一种特殊情况，将会在完成动态壁纸的学习之后于8.3节进行讨论。而WindowManagerService对壁纸窗口的处理将在8.4节进行介绍。

8.2 深入理解动态壁纸

8.2.1 启动动态壁纸的方法

启动动态壁纸可以通过调用WallpaperManager.getIWallpaperManager().setWallpaperComponent()方法完成。它接受一个ComponentName类型的参数，用于将希望启动的壁纸的WallpaperService的

ComponentName告知WallpaperManagerService。WallpaperManager.getIWallpaperManager()方法返回的是WallpaperManagerService的Bp端。因此setWallpaperComponent()方法的实现位于WallpaperManagerService之中。参考其实现：

```
[WallpaperManagerService.java-->WallpaperManagerService.setWallpaperComponent()]
public void setWallpaperComponent(ComponentNamename) {
    // 设置动态壁纸需要调用者拥有一个签名级的系统权限
    checkPermission(android.Manifest.permission.SET_WALLPAPER_COMPONENT);
   synchronized (mLock) {
        /* **① 首先从mWallpaperMap中获取壁纸的运行信息WallpaperData。**
         WallpaperManagerService支持多用户机制，因此设备上的每一个用户可以设置自己
          的壁纸。mWallpaperMap中为每一个用户保存了一个WallpaperData实例，这个实例
          中保存了和壁纸运行状态相关的信息。例如WallpaperService的ComponentName，
          到WallpaperService的ServiceConnection等。于是当发生用户切换时，
          WallpaperManagerService可以从mWallpaperMap中获取新用户的WallpaperData，
          并通过保存在其中的ComponentName重新启动该用户所设置的壁纸。因此，
          当通过setWallpaperComponent()设置新壁纸时，需要获取当前用户的WallpaperData，
          并在随后更新其内容使之保存新壁纸的信息 */
        intuserId = UserHandle.getCallingUserId();
       WallpaperData wallpaper = mWallpaperMap.get(userId);
       ......
       final long ident = Binder.clearCallingIdentity();
        try{
           ......
           // **② 启动新壁纸的WallpaperService**
           bindWallpaperComponentLocked(name, false, true, wallpaper, null);
        }finally {
           Binder.restoreCallingIdentity(ident);
        }
    }
}
```

注意 WallpaperManager.getIWallpaperManager()并没有作为SDK的一部分提供给开发者。因此第三方应用程序是无法进行动态壁纸的设置的。

8.2.2 壁纸服务的启动原理

（1）壁纸服务的验证与启动

bindWallpaperComponentLocked()方法将会启动由ComponentName所指定的WallpaperService，并向WMS申请用于添加壁纸窗口的窗口令牌。不过在此之前，bindWallpaperComponentLocked()会对ComponentName所描述的Service进行一系列的验证，以确保它是一个壁纸服务。而这一系列的验证过程体现了一个Android服务可以被当作壁纸必要的条件。

```
[WallpaperManagerService.java-->WallpaperManagerService.setWallpaperComponentLocked()]
boolean bindWallpaperComponentLocked(ComponentNamecomponentName, boolean force,
       boolean fromUser, WallpaperData wallpaper, IRemoteCallback reply) {
    ......
    try {
        /* 当componentName为null时表示使用默认壁纸。
          这里会将componentName参数改为默认壁纸的componentName */
        if(componentName == null) {
           /* 首先会尝试从com.android.internal.R.string.default_wallpaper_component
              中获取默认壁纸的componentName。这个值的设置位于res/values/config.xml中，
              开发者可以通过修改这个值设置默认壁纸*/
```

```
            String defaultComponent = mContext.getString(
                          com.android.internal.R.string.default_wallpaper_component);
            if (defaultComponent != null) {
                componentName = ComponentName.unflattenFromString(defaultComponent);
            }
            /* 倘若在上述的资源文件中没有指定一个默认壁纸，即default_wallpaper_component的
               值被设置为@null)，则使用ImageWallpaper代替默认壁纸。ImageWallpaper就是前文
               所述的静态壁纸 */
            if (componentName == null) {
                componentName = IMAGE_WALLPAPER;
            }
        }
        /* 接下来WallpaperMangerService会尝试从PackageManager中尝试获取ComponentName所
          指定的Service的描述信息,获取此信息的目的在于确认该Service是一个符合要求的壁纸服务 */
        intserviceUserId = wallpaper.userId;
       ServiceInfo si = mIPackageManager.getServiceInfo(componentName,
                             PackageManager.GET_META_DATA |
                              PackageManager.GET_PERMISSIONS,serviceUserId);
        /* **① 第一个检查，要求这个Service必须声明其访问权限为BIND_WALLPAPER。**这个签名级的系
          统权限这是为了防止壁纸服务被第三方应用程序启动而产生混乱 */
        if(!android.Manifest.permission.BIND_WALLPAPER.equals(si.permission)) {
           if (fromUser) {
               throw new SecurityException(msg);
           }
           return false;
        }
       WallpaperInfo wi = null;
        /* **② 第二个检查，要求这个Service必须可以用来处理**
**            android.service.wallpaper.WallpaperService这个Action。**
           其检查方式是从PackageManager中查询所有可以处理
           android.service.wallpaper.WallpaperService的服务，然后检查即将启动的服务
           是否在PackageManager的查询结果之中 */
       Intent intent = new Intent(WallpaperService.SERVICE_INTERFACE);
        if(componentName != null && !componentName.equals(IMAGE_WALLPAPER)) {
           // 获取所有可以处理android.service.wallpaper.WallpaperService的服务信息
           List&lt;ResolveInfo&gt; ris =
                   mIPackageManager.queryIntentServices(intent,
                           intent.resolveTypeIfNeeded(mContext.getContentResolver()),
                           PackageManager.GET_META_DATA, serviceUserId);
            /* **③ 第三个检查，要求这个Service必须在其meta-data中提供关于壁纸的描述信息。**如果
             即将启动的服务位于查询结果之中，便可以确定这是一个壁纸服务。此时会创建一
             个WallpaperInfo的实例以解析并存储此壁纸服务的描述信息。壁纸服务的描述信息包含
             了壁纸的开发者、缩略图、简单的描述文字以及用于对此壁纸进行参数设置的Activity的
             名字等。壁纸开发者可以在AndroidManifest.xml中将一个包含了上述信息的xml文件设
             置在名为android.service.wallpaper的meta-data中以提供这些信息 */
           for (int i=0; i&lt;ris.size(); i++) {
               ServiceInfo rsi = ris.get(i).serviceInfo;
               if (rsi.name.equals(si.name) &&
                        rsi.packageName.equals(si.packageName)){
                   try {
                        wi = newWallpaperInfo(mContext, ris.get(i));
                   } catch (XmlPullParserException e) {......}
                   break;
               }
           }
            if (wi == null) {
               /* wi为null表示即将启动的服务没有位于查询结果之中，或者没有提供必须的meta-data。
                 此时返回false表示绑定失败 */
               return false;
           }
        }
       ......
    }
    ......
}
```

可见WallpaperManagerService要求被启动的目标Service必须满足以下三个条件：

- 该服务必须要以android.permission.BIND_WALLPAPER作为其访问权限。壁纸虽然是一个标准的Android服务，但是通过其他途径（如第三方应用程序）启动壁纸所在的服务是没有意义的。因此Android要求作为壁纸的Service必须使用这个签名级的系统权限进行访问限制，以免被意外的应用程序启动。

- 该服务必须被声明为可以处理android.service.wallpaper.WallpaperService这个Action。WallpaperManagerService会使用这个Action对此服务进行绑定。

- 该服务必须在其AndroidManifest.xml中提供一个名为android.service.wallpaper的meta-data，用于提供动态壁纸的开发者、缩略图与描述文字。

一旦目标服务满足了上述条件，WallpaperManagerService就会着手进行目标服务的启动与绑定。

参考setWallpaperComponentLocked()方法的后续代码：

```
[WallpaperManagerService.java-->WallpaperManagerService.setWallpaperComponentLocked()]
boolean bindWallpaperComponentLocked(ComponentNamecomponentName, boolean force,
       boolean fromUser, WallpaperData wallpaper, IRemoteCallback reply) {
       ...... // 检查服务是否符合要求的代码
        /* **① 创建一个WallpaperConnection。**它不仅实现了ServiceConnection接口用于监
          听和WallpaperService之间的连接状态，同时还实现了IWallpaperConnection.Stub，
          也就是说它支持跨进程通信。
          在服务绑定成功后的WallpaperConnection.onServiceConnected()方法调用中，
          WallpaperConnection的实例会被发送给WallpaperService，使其作为WallpaperService
          向WallpaperManagerService进行通信的桥梁 */
       WallpaperConnection newConn = new WallpaperConnection(wi, wallpaper);
        // 为启动壁纸服务准备Intent
       intent.setComponent(componentName);
       intent.putExtra(Intent.EXTRA_CLIENT_LABEL,
               com.android.internal.R.string.wallpaper_binding_label);
       intent.putExtra(Intent.EXTRA_CLIENT_INTENT,PendingIntent.getActivityAsUser(
               mContext, 0,
               Intent.createChooser(new Intent(Intent.ACTION_SET_WALLPAPER),
                 mContext.getText(com.android.internal.R.string.chooser_wallpaper)),
               0, null, new UserHandle(serviceUserId)));
        /* **② 启动制定的壁纸服务。**当服务启动完成后，剩下的启动流程会在
          WallpaperConnection.onServiceConnected()中继续 */
        if(!mContext.bindService(intent,
                              newConn,Context.BIND_AUTO_CREATE, serviceUserId)) {
        }
        // **③ 新的的壁纸服务启动成功后，便通过detachWallpaperLocked()销毁旧有的壁纸服务**
        if(wallpaper.userId == mCurrentUserId && mLastWallpaper != null) {
           detachWallpaperLocked(mLastWallpaper);
        }
        // **④ 将新的壁纸服务的运行信息保存到WallpaperData中**
       wallpaper.wallpaperComponent = componentName;
       wallpaper.connection = newConn;
        /* 设置wallpaper.lastDiedTime。这个成员变量与其说描述壁纸的死亡时间戳，不如说是
          描述其启动的时间戳。它用来在壁纸服务意外断开时(即壁纸服务非正常停止) 检查此壁纸服务
          的存活时间。当存活时间小于一个特定的时长时将会认为这个壁纸的软件质量不可靠
          从而选择使用默认壁纸，而不是重启这个壁纸服务 */
       wallpaper.lastDiedTime = SystemClock.uptimeMillis();
       newConn.mReply = reply;
        /* **④ 最后向WMS申请注册一个WALLPAPER类型的窗口令牌。**这个令牌会在onServiceConnected()
          之后被传递给WallpaperService用于作为后者添加窗口的通行证 */
        try{
           if (wallpaper.userId == mCurrentUserId) {
                mIWindowManager.addWindowToken(newConn.mToken,
                       WindowManager.LayoutParams.TYPE_WALLPAPER);
               mLastWallpaper = wallpaper;
           }
        } catch (RemoteException e) {}
    } catch(RemoteException e) {}
    returntrue;
}
```

bindWallpaperComponentLocked()主要做了如下几件事情：

- 创建WallpaperConnection。由于实现了ServiceConnection接口，因此它将负责监听WallpaperManagerService与壁纸服务之间的连接状态。另外由于继承了IWallpaperConnection.Stub，因此它具有跨进程通信的能力。在壁纸服务绑定成功后，WallpaperConnection实例会被传递给壁纸服务作为壁纸服务与WallpaperManagerService进行通信的桥梁。
- 启动壁纸服务。通过Context.bindService()方法完成。可见启动壁纸服务与启动一个普通的服务没有什么区别。

- 终止旧有的壁纸服务。
- 将属于当前壁纸的WallpaperConnection实例、componentName机器启动时间戳保存到WallpaperData中。
- 向WMS注册WALLPAPER类型的窗口令牌。这个窗口令牌保存在WallpaperConnection.mToken中，并随着WallpaperConnection的创建而创建。

仅仅将指定的壁纸服务启动起来尚无法使得壁纸得以显示，因为新启动起来的壁纸服务由于没有可用的窗口令牌而导致其无法添加窗口。WallpaperManagerService必须通过某种方法将窗口令牌交给壁纸服务才行。所以壁纸显示的后半部分的流程将在WallpaperConnection.onServiceConnected()回调中继续。同其他服务一样，WallpaperManagerService会在这个回调之中获得一个Binder对象。因此在进行onServiceConnected()方法的讨论之前，必须了解WallpaperManagerService在这个回调中将会得到一个什么样的Binder对象。

现在把分析目标转移到WallpaperService中。和普通服务一样，WallpaperService的启动也会经历onCreate()、onBind()这样的生命周期回调。为了了解WallpaperManagerService可以从onServiceConnected()获取怎样的Binder对象，需要看下WallpaperService.onBind()的实现：

```
[WallpaperService.java-->WallpaperService.onBind()]
public final IBinder onBind(Intent intent) {
    /*onBind()新建了一个IWallpaperServiceWrapper实例，并将
      其返回给WallpaperManagerService */
    returnnew IWallpaperServiceWrapper(this);
}
```

IWallpaperServiceWrapper类继承自IWallpaperService.Stub。它保存了WallpaperService的实例，同时也实现了唯一的一个接口attach()。很显然，当这个Binder对象返回给WallpaperManagerService之后，后者定会调用这个唯一的接口attach()以传递显示壁纸所必须的包括窗口令牌在内的一系列的参数。

（2）向壁纸服务传递创建窗口所需的信息

重新回到WallpaperManagerService，当WallpaperService创建了IWallpaperServiceWrapper实例并返回后，WallpaperManagerService将会在WallpaperConnection.onServiceConnected()中收到回调。参考其实现：

```
[WallpaperManagerService.java-->WallpaperConnection.onServiceConnected()]
public void onServiceConnected(ComponentName name,IBinder service) {
   synchronized (mLock) {
        if (mWallpaper.connection == this) {
           // 更新壁纸的启动时间戳
           mWallpaper.lastDiedTime = SystemClock.uptimeMillis();
           // **① 将WallpaperService传回的IWallpaperService接口保存为mService**
           mService = IWallpaperService.Stub.asInterface(service);
           /* **② 绑定壁纸服务。**attachServiceLocked()会调用IWallpaperService.attach()
             方法以将壁纸服务创建窗口所需的信息传递过去 */
           attachServiceLocked(this, mWallpaper);
           // **③ 保存当前壁纸的运行状态到文件系统中，以便在系统重启或发生用户切换时可以恢复**
           saveSettingsLocked(mWallpaper);
        }
    }
}
进一步地，attachServiceLocked()方法会调用IWallpaperService.attach()方法，将创建壁纸窗口所需的信息
[WallpaperManagerService.java-->WallpaperManagerService.attachServiceCLocked()]
void attachServiceLocked(WallpaperConnection conn,WallpaperData wallpaper) {
    try {
        /* 调用IWallpaperService的唯一接口attach()，将创建壁纸窗口所需要的参数传递
          给WallpaperService */
       conn.mService.attach(conn, conn.mToken,
               WindowManager.LayoutParams.TYPE_WALLPAPER, false,
               wallpaper.width, wallpaper.height);
    } catch(RemoteException e) {......}
}
```

attach()方法的参数很多，它们的意义如下:

- conn即WallpaperConnection，WallpaperService将通过它向WallpaperManagerService进行通信。WallpaperConnection继承自IWallpaperConnection，只提供了两个接口的定义，即attachEngine()以及engineShown()。虽说WallpaperManager是WallpaperManagerService向外界提供的标准接口，但是这里仍然选择使用WallpaperConnection实现这两个接口的原因是由于attachEngine()以及engineShown()是只有WallpaperService才需要用到而且是它与 WallpaperManagerService之间比较底层且私密的交流，将它们的实现放在通用的接口WallpaperManager中显然并不合适。这两个接口中比较重要的当属attachEngine()了。如前文所述，Engine类是实现壁纸的核心所在，而WallpaperService只是一个用于承载壁纸的运行的容器而已。因此相对于WallpaperService，Engine是WallpaperManagerService更加关心的对象。所以当WallpaperService完成了Engine对象的创建之后，就会通过attachEngine()方法将Engine对象的引用交给WallpaperManagerService。

- conn.mToken就是在bindWallpaperComponent()方法中向WMS注册过的窗口令牌。是WallpaperService有权添加壁纸窗口的凭证。

- WindowManager.LayoutParams.TYPE_WALLPAPER指明了WallpaperService需要添加TYPE_WALLPAPER类型的窗口。读者可能会质疑这个参数的意义：壁纸除了是TYPE_WALLPAPER类型以外难道还有其他的可能么？的确在实际的壁纸显示中WallpaperService必然需要使用TYPE_WALLPAPER类型添加窗口。但是有一个例外，即壁纸预览。在LivePicker应用中选择一个动态壁纸时，首先会使得用户对选定的壁纸进行预览。这一预览并不是真的将壁纸设置给了WallpaperManagerService，而是

LivePicker应用自行启动了对应的壁纸服务，并要求壁纸服务使用TYPE_APPLICATION_MEDIA_OVERLAY类型创建窗口。这样一来,壁纸服务所创建的窗口将会以子窗口的形式衬在LivePicker的窗口之下，从而实现了动态壁纸的预览。

- false的参数名是isPreview. 用以指示启动壁纸服务的意图。当被实际用作壁纸时取值为false，而作为预览时则为true。仅当LivePicker对壁纸进行预览时才会使用true作为isPreview的取值。壁纸服务可以根据这一参数的取值对自己的行为作出调整。

当WallpaperManagerService向WallpaperService提供了用于创建壁纸窗口的足够的信息之后，WallpaperService便可以开始着手进行Engine对象的创建了。

（3）Engine的创建

调用IWallpaperService.attach()是WallpaperManagerService在壁纸服务启动后第一次与壁纸服务进行联系。参考其实现：

```
[WallpaperService.java-->IWallpaperServiceWrapper.attach()]
public void attach(IWallpaperConnection conn,IBinder windowToken,
        intwindowType, boolean isPreview, int reqWidth, int reqHeight) {
    // 使用WallpaperManagerService提供的参数构造一个IWallpaperEngineWarapper实例
    newIWallpaperEngineWrapper(mTarget, conn, windowToken,
            windowType, isPreview, reqWidth, reqHeight);
}
```

顾名思义，在attach()方法中所创建的IWallpaperEngineWrapper将会创建并封装Engine实例。IWallpaperEngineWrapper继承自IWallpaperEngine.Stub，因此它也支持跨Binder调用。在随后的代码分析中可知，它将会被传递给WallpaperManagerService，作为WallpaperManagerService与Engine进行通信的桥梁。

另外需要注意的是，attach()方法的实现非常奇怪，它直接创建一个实例但是并没有将这个实例赋值给某一个成员变量，在attach()方法结束时岂不是会被垃圾回收？不难想到，在IWallpaperEngineWrapper的构造函数中一定有些动作可以使得这个实例不被释放。参考其实现：

```
[WallpaperService.java-->IWallpaperEngineWrapper.IWallpaperEngineWrapper()]
IWallpaperEngineWrapper(WallpaperService context,
       IWallpaperConnection conn, IBinder windowToken,
        intwindowType, boolean isPreview, int reqWidth, int reqHeight) {
    /* 创建一个HandlerCaller。
      HandlerCaller是Handler的一个封装，而它与Handler的区别是额外提供了
      一个executeOrSendMessage()方法。当开发者在HandlerCaller所在的线程
      执行此方法时会使得消息的处理函数立刻得到执行，在其他线程中执行此方法的效果
      则与Handler.sendMessage()别无二致。除非阅读代码时遇到这个方法，读者
      只需要将其理解为Handler即可。
      注意意通过其构造函数的参数可知HandlerCaller保存了IWallpaperEngineWrapper的实例 */
    mCaller= new HandlerCaller(context,
           mCallbackLooper != null
                   ? mCallbackLooper : context.getMainLooper(),
           this);
    // 将WallpaperManagerService所提供的参数保存下来
   mConnection = conn; // conn即是WallpaperManagerService中的WallpaperConnection
   mWindowToken = windowToken;
   mWindowType = windowType;
   mIsPreview = isPreview;
   mReqWidth = reqWidth;
   mReqHeight = reqHeight;
    // 发送DO_ATTACH消息。后续的流程转到DO_ATTACH消息的处理中进行
    Messagemsg = mCaller.obtainMessage(DO_ATTACH);
   mCaller.sendMessage(msg);
}
```

注意 在这里貌似并没有保存新建的IWallpaperEngineWrapper实例，它岂不是有可能在DO_ATTACH消息执行前就被Java的垃圾回收机制回收了？其实不是这样。HandlerCaller的构造函数以及最后的sendMessage()操作使得这个IWallpaperEngineWrapper的实例得以坚持到DO_ATTACH消息可以得到处理的时刻。sendMessage()方法的调用使得Message被目标线程的MessageQueue引用，并且对应的Handler的被Message引用，而这个Handler是HandlerCaller的内部类，因此在Handler中有一个隐式的指向HandlerCaller的引用，最后在HandlerCaller中又存在着IWallpaperEngineWrapper的引用。因此IWallpaperEngineWrapper间接地被HandlerCaller所在线程的MessageQueue所引用着，因此在完成DO_ATTACH消息的处理之前，IWallpaperEngineWrapper并不会被回收。虽然这是建立在对Java引用以及Handler工作原理的深刻理解之上所完成的精妙实现，但是它确实已经接近危险的边缘了。

在这里所创建的mCaller具有十分重要的地位。它是一个重要的线程调度器，所有壁纸相关的操作都会以消息的形式发送给mCaller，然后在IWallpaperEngineWrapper的executeMessage()方法中得到处理，从而这些操作转移到mCaller所在的线程上进行（如壁纸绘制、事件处理等）。可以说mCaller的线程就是壁纸的工作线程。默认情况下这个mCaller运行在壁纸服务的主线程上即context.getMainLooper()。不过当WallpaperService.mCallbackLooper不为null时会运行在mCallbackLooper所在的线程。mCaller运行在壁纸服务的主线程上听起来十分合理，然而提供手段以允许其运行在其他线程的做法却有些意外。其实这是为了满足一种特殊的需求，以ImageWallper壁纸服务为例，它是SystemUI的一部分而SystemUI的主线程主要用来作为状态栏、导航栏的管理与绘制的场所，换句话说其主线程的工作已经比较繁重了。因此ImageWallpaper可以通过这一手段将壁纸的工作转移到另外一个线程中进行。不过因为这一机制可能带来同步上的问题，因此在Android 4.4及后续版本中被废除了。

接下来分析DO_ATTACH消息的处理：

```
[WallpaperService.java-->IWallpaperEngineWrapper.executeMessage()]
public void executeMessage(Message message) {
    switch(message.what) {
        caseDO_ATTACH: {
           try {
               /* **① 把IWallpaperEngineWrapper实例传递给WallpaperConnection进行保存。**
                至此这个实例便名花有主，再也不用担心被回收了，而且WallpaperManagerService
                还可以通过它与实际的Engine进行通信 */
               mConnection.attachEngine(this);
           } catch (RemoteException e) {}
           /* **② 通过onCreateEngine()方法创建一个Engine。**
             onCreateEngine()是定义在WallpaperService中的一个抽象方法。
             WallpaperService的实现者需要根据自己的需要返回一个自定义的Engine的子类 */
           Engine engine = onCreateEngine();
           mEngine = engine;
           /* **③ 将新建的Engine添加到WallpaperService.mActiveEngines列表中。**
             读者可能会比较奇怪，为什么会是列表？难道一个Wallpaper可能会有多个Engine么？
            这个奇怪之处还是是壁纸预览所引入的。当壁纸A已经被设置为当前壁纸之时，系统中会存
            在一个它所对应的WallpaperService,以及在其内部会存在一个Engine。
            此时当LivePicker或其他壁纸管理工具预览壁纸A时，它所对应的WallpaperService
            仍然只有一个，但是在其内部会变成两个Engine。
            这一现象更能说明，WallpaperService仅仅是提供壁纸运行的场所，而Engine才是真正
            的壁纸的实现 */
           mActiveEngines.add(engine);
           // **④ 最后engine.attach()将会完成窗口的创建、第一帧的绘制等工作**
            engine.attach(this);
           return;
        }
    }
}
```

正如前文所述，作为拥有跨Binder调用的IWallpaperEngineWrapper通过attachEngine()方法将自己传递给了WallpaperConnection，后者将其保存在WallpaperConnection.mEngine成员之中。从此之后，WallpaperManagerService便可以通过WallpaperConnection.mEngine与壁纸服务进程中的IWallpaperEngineWrapper进行通信，而IWallpaperEngineWrapper进一步将来自WallpaperManagerService中的请求或设置转发给Engine对象，从而实现了WallpaperManagerService对壁纸的控制。

到目前为止，WallpaperManagerService与壁纸服务之间已经出现了三个用于跨Binder通信的对象。它们分别是：

- IWallpaperService，实现在壁纸服务进程之中，它所提供的唯一的方法attach()用于在壁纸服务启动后接收窗口创建所需的信息，或者说为了完成壁纸的初始化工作。除此之外IWallpaperService不负责任何功能，WallpaperManagerService对壁纸进行的请求与设置都交由在attach()的过程中所创建的IWallpaperEngineWrapper实例完成。

- WallpaperConnection，实现在WallpaperManagerService中，并通过IWallpaperService.attach()方法传递给了IWallpaperEngineWrapper。壁纸服务通过WallpaperConnection的attachEngine()方法将IWallpaperEngineWrapper实例传递给WallpaperManagerService进行保存。另外壁纸服务还通过它的engineShown()方法将壁纸显示完成的事件通知给WallpaperManagerService。

- IWallpaperEngineWrapper，实现在壁纸进程中。Engine实例是壁纸实现的核心所在。作为Engine实例的封装者，它是WallpaperManagerService对Engine进行请求或设置的唯一接口。

总体来说，IWallpaperService与WallpaperConnection主要服务于壁纸的创建阶段，而IWallpaperEngineWrapper则用于在壁纸的运行阶段对Engine进行操作与设置。

说明 按照常规的思想来推断，WallpaperManagerService与WallpaperService之间应该仅仅需要IWallpaperService提供接口对壁纸进行操作与设置。为什么要增加一个IWallpaperEngineWrapper呢？这得从WallpaperService与Engine之间的关系说起。IWallpaperService在WallpaperManagerService看来表示的是WallpaperService，而IWallpaperEngineWrapper则表示的是Engine。WallpaperService是Engine运行的容器，因此它所提供的唯一的方法attach()用来在WallpaperService中创建新的Engine实例（由创建一个IWallpaperEngineWrapper实例来完成）。Engine则是壁纸的具体实现，因此IWallpaperEngineWrapper所提供的方法用来对壁纸进行操作与设置。从这个意义上来讲IWallpaperService与IWallpaperEngineWrapper的同时存在是合理的。另外，将IWallpaperService与IWallpaperEngineWrapper分开还有着简化实现的意义。从DO_ATTACH消息的处理过程可知，WallpaperService中可以同时运行多个Engine实例。而WallpaperManagerService或LivePicker所关心的只是某一个Engine，而不是WallpaperService中的所有Engine，因此相对于使用IWallpaperService的接口时必须在参数中指明所需要操作的Engine，直接操作IWallpaperEngineWrapper更加简洁直接。

Engine创建完毕之后会通过Engine.attach()方法完成Engine的初始化工作。参考其代码：

```
[WallpaperService.java-->Engine.attach()]
void attach(IWallpaperEngineWrapper wrapper) {
    ......
    // 保存必要的信息
   mIWallpaperEngine = wrapper;
    mCaller= wrapper.mCaller;
   mConnection = wrapper.mConnection;
   mWindowToken = wrapper.mWindowToken;
    /* **① mSurfaceHolder是一个BaseSurfaceHolder类型的内部类的实例。**
      Engine对其进行了简单的定制。开发者可以通过mSurfaceHolder定制所需要的Surface类型 */
   mSurfaceHolder.setSizeFromLayout();
   mInitializing = true;
    // 获取WindowSession,用于与WMS进行通信
    mSession= WindowManagerGlobal.getWindowSession(getMainLooper());
    //mWindow是IWindow的实现，窗口创建之后它将用于接收来自WMS的回调
   mWindow.setSession(mSession);
    //Engine需要监听屏幕状态。这是为了保证在屏幕关闭之后，动态壁纸可以停止动画的渲染以节省电量
   mScreenOn =
           ((PowerManager)getSystemService(Context.POWER_SERVICE)).isScreenOn();
   IntentFilter filter = new IntentFilter();
    filter.addAction(Intent.ACTION_SCREEN_ON);
   filter.addAction(Intent.ACTION_SCREEN_OFF);
   registerReceiver(mReceiver, filter);
    /* **② 调用Engine.onCreate()。**
      Engine的子类往往需要重写此方法以修改mSurfaceHolder的属性，如像素格式，尺寸等。
      注意此时尚未创建窗口，在这里所设置的SurfaceHolder的属性将会在创建窗口时生效 */
   onCreate(mSurfaceHolder);
   mInitializing = false;
   mReportedVisible = false;
    /* **③ 最后updateSurface将会根据SurfaceHolder的属性创建窗口以及Surface，并进行**
**       壁纸的第一次绘制** */
   updateSurface(false, false, false);
}
```

Engine.attach()方法执行的结束标志着壁纸启动工作的完成，至此在最后的updateSurface()方法结束之后新的壁纸便显示出来了。

（4）壁纸的创建流程

可见，壁纸的创建过程比较复杂。在这个过程中存在着多个Binder对象之间的互相调用。因此有必要对此过程进行一个简单的整理：

- 首先，壁纸管理程序（如LivePicker)调用IWallpaperManager.setWallpaperComponent()要求WallpaperManagerService设置指定的壁纸
- WallpaperManagerService通过调用bindWallpaperComponentLocked()将给定的壁纸服务启动起来。同时旧有的壁纸服务会被终止。
- WallpaperManagerService成功连接壁纸服务后，调用壁纸服务的attach()方法将窗口令牌等参数交给壁纸服务。
- 壁纸服务响应attach()的调用，创建一个Engine。
- Engine的updateSurface()方法将会创建壁纸窗口及Surface，并进行壁纸的绘制。

而在这个过程中，WallpaperManagerService中存在如下重要的数据结构：

- WallpaperInfo，存储了动态壁纸的开发者、缩略图与描述信息。这个数据结构创建于WallpaperManagerService.bindWallpaperComponentLocked()方法，其内容来自于壁纸所在应用程序的AndroidManifest.xml中名为android.service.wallpaper的meta-data。
- WallpaperConnection，它不仅仅是壁纸服务与WallpaperManagerService进行通信的渠道，它同时也保存了与壁纸服务相关的重要的运行时信息，如IWallpaperService、IWallpaperEngineWrapper、WallpaperInfo以及用于创建窗口所需的窗口令牌。WallpaperConnection创建于WallpaperManagerService.bindWallpaperComponentLocked()方法。
- WallpaperData，它保存了一个壁纸在WallpaperManagerService中可能用到的所有信息，包括壁纸服务的ComponentName，WallpaperConnection，壁纸服务的启动时间等。WallpaperData被保存在一个名为mWallpaperMap的SparseArray中，而且设备中每一个用户都会拥有一个固定的WallpaperData实例。当前用户进行壁纸切换时会更新WallpaperData的内容，而不是新建一个WallpaperData实例。另外，WallpaperData中还保存了与静态壁纸相关的一些信息，关于静态壁纸的内容将在8.3节进行介绍。

壁纸的创建过程同时体现了壁纸服务与WallpaperManagerService之间的关系，如图8-1所示。

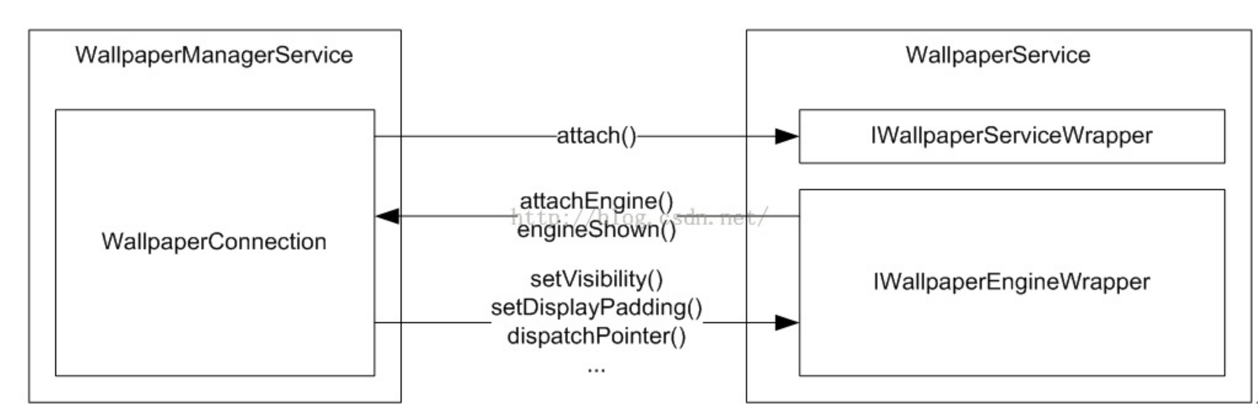


图 8 - 1 壁纸服务与WallpaperManagerService之间的关系